Messaging Layer Security
Internet-Draft
Intended status: Standards Track
Expires: 22 January 2026

R. Robert
Phoenix R&D

21 July 2025

The Messaging Layer Security (MLS) Extensions
draft-ietf-mls-extensions-08

Abstract

The Messaging Layer Security (MLS) protocol is an asynchronous group authenticated key exchange protocol. MLS provides a number of capabilities to applications, as well as several extension points internal to the protocol. This document provides a consolidated application API, guidance for how the protocol's extension points should be used, and a few concrete examples of both core protocol extensions and uses of the application API.

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://mlswg.github.io/mls-extensions/draft-ietf-mls-extensions.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-mls-extensions/.

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/mlswg/mls-extensions.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document defines extensions to MLS [RFC9420] that are not part of the main protocol specification, and uses them to explain how to extend the core operation of the MLS protocol. It also describes how applications can safely interact with MLS to take advantage of security features of MLS.

The MLS protocol is designed to be integrated into applications in order to provide security services that the application requires. There are two questions to answer when designing such an integration:

1. How does the application provide the services that MLS requires?

2. How does the application use MLS to get security benefits?

The MLS Architecture [I-D.ietf-mls-architecture] describes the requirements for the first of these questions, namely the structure of the Delivery Service and Authentication Service that MLS requires. The next section of this document focuses on the second question.

MLS itself offers some basic functions that applications can use, such as the secure message encapsulation (PrivateMessage), the MLS exporter, and the epoch authenticator. Current MLS applications make use of these mechanisms to achieve a variety of confidentiality and authentication properties.

As application designers become familiar with MLS, there is interest in leveraging other cryptographic tools that an MLS group provides:

* HPKE (Hybrid Public Key Encryption [RFC9180]) and signature key pairs for each member, where the private key is known only to that member, and the public key is authenticated to the other members.

* A pre-shared key mechanism that can allow an application to inject data into the MLS key schedule.

* An exporter mechanism that allows applications to derive secrets from the MLS key schedule.

* Association of data with Commits as a synchronization mechanism.

* Binding of information to the GroupContext to confirm group agreement.

There is also interest in exposing an MLS group to multiple loosely coordinated components of an application. To accommodate such cases, the above mechanisms need to be exposed in such a way that the usage of different components do not conflict with each other, or with MLS itself.

This document defines a set of mechanisms that application components can use to ensure that their use of these facilities is properly domain-separated from MLS itself, and from other application components that might be using the same MLS group.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document makes heavy use of the terminology and the names of structs in the MLS specification [RFC9420]. In addition, we introduce the following new terms:

Application: The system that instantiates, manages, and uses an MLS group. Each MLS group is used by exactly one application, but an application may maintain multiple groups.

Application component: A subsystem of an application that has access to an MLS group.

Component ID: An identifier for an application component. These identifiers are assigned by the application.

3. Developing Extensions for the MLS Protocol

MLS is highly extensible and was designed to be used in a variety of different applications. As such, it is important to separate extensions that change the behavior of an MLS stack, and application usages of MLS that just take advantage of features of MLS or specific extensions (hereafter referred to as _components_). Furthermore, it is essential that application components do not change the security properties of MLS or require new security analysis of the MLS protocol itself.

4. The Safe Application Interface

The mechanisms in this section take MLS mechanisms that are either not inherently designed to be used by applications, or not inherently designed to be used by multiple application components, and adds a domain separator that separates application usage from MLS usage, and application components' usage from each other:

* Public-key encryption operations are tagged so that encrypted data will only decrypt in the context of a given component.

* Signing operations are similarly tagged so that signatures will only verify in the context of a given component.

* Exported values include an identifier for the component to which they are being exported, so that different components will get different exported values.

* Pre-shared keys are identified as originating from a specific component, so that different components' contributions to the MLS key schedule will not collide.

* Additional Authenticated Data (AAD) can be domain separated by component.

Similarly, the content of application messages (application_data) can be distinguished and routed to different parts of an application according to the media type of that content using the content negotiation mechanism defined in Section 6.2.

We also define new general mechanisms that allow applications to take advantage of the extensibility mechanisms of MLS without having to define extensions themselves:

* An app_data_dictionary extension type that associates application data with MLS messages or with the state of the group.

* An AppEphemeral proposal type that enables arbitrary application data to be associated with a Commit.

* An AppDataUpdate proposal type that enables efficient updates to an app_data_dictionary GroupContext extension.

As with the above, information carried in these proposals and extensions is marked as belonging to a specific application component, so that components can manage their information independently.

The separation between components is achieved by the application assigning each component a unique component ID number. These numbers are then incorporated into the appropriate calculations in the protocol to achieve the required separation.

4.1. Component IDs

A component ID is a four-byte value that uniquely identifies a component within the scope of an application.

```
uint32 ComponentID;
```

When a label is required for an operation, the following data structure is used. The base_label field is always the fixed string "Application". The component_id field identifies the component performing the operation. The label field identifies the operation being performed.

```
struct {
  opaque base_label<V>; /* = "Application" */
  ComponentID component_id;
  opaque label<V>;
} ComponentOperationLabel;
```

4.2. Hybrid Public Key Encryption (HPKE) Keys

This component of the API allows components to make use of the HPKE key pairs generated by MLS. A component identified by a ComponentID can use any HPKE key pair for any operation defined in [RFC9180], such as encryption, exporting keys, and the PSK mode, as long as the info input to Setup<MODE>S and Setup<MODE>R is set to ComponentOperationLabel with component_id set to the appropriate ComponentID. The context can be set to an arbitrary Context specified by the application designer and can be empty if not needed. For example, a component can use a key pair PublicKey, PrivateKey to encrypt data as follows:

```
SafeEncryptWithLabel(PublicKey, ComponentID, Label, Context, Plaintext) =
  EncryptWithLabel(PublicKey, ComponentOperationLabel, Context, Plaintext)

SafeDecryptWithLabel(PrivateKey, ComponentID, Label, Context, KEMOutput,
  Ciphertext) =
  DecryptWithLabel(PrivateKey, ComponentOperationLabel, Context,
    KEMOutput, Ciphertext)
```

Where the fields of ComponentOperationLabel are set to

```
component_id = ComponentID
label = Label
```

For operations involving the secret key, ComponentID MUST be set to the ComponentID of the component performing the operation, and not to the ID of any other component. In particular, this means that a component cannot decrypt data meant for another component, while components can encrypt data that other components can decrypt.

In general, a ciphertext encrypted with a PublicKey can be decrypted by any entity who has the corresponding PrivateKey at a given point in time according to the MLS protocol (or application component). For convenience, the following list summarizes lifetimes of MLS key pairs.

* The key pair of a non-blank ratchet tree node. The PrivateKey of such a key pair is known to all members in the node's subtree. In particular, a PrivateKey of a leaf node is known only to the member in that leaf. A member in the subtree stores the PrivateKey for a number of epochs, as long as the PublicKey does not change. The key pair of the root node SHOULD NOT be used, since the external key pair recalled below gives better security.

* The external_priv, external_pub key pair used for external initialization. The external_priv key is known to all group members in the current epoch. A member stores external_priv only for the current epoch. Using this key pair gives better security guarantees than using the key pair of the root of the ratchet tree and should always be preferred.

* The init_key in a KeyPackage and the corresponding secret key. The secret key is known only to the owner of the KeyPackage and is deleted immediately after it is used to join a group.

4.3. Signature Keys

MLS session states contain a number of signature keys including the ones in the LeafNode structs. Application components can safely sign content and verify signatures using these keys via the SafeSignWithLabel and SafeVerifyWithLabel functions, respectively, much like how the basic MLS protocol uses SignWithLabel and VerifyWithLabel.

In more detail, a component identified by ComponentID should sign and verify using:

```
SafeSignWithLabel(SignatureKey, ComponentID, Label, Content) =
  SignWithLabel(SignatureKey, ComponentOperationLabel, Content)

SafeVerifyWithLabel(VerificationKey, ComponentID, Label, Content,
  SignatureValue) =
  VerifyWithLabel(VerificationKey, ComponentOperationLabel, Content,
  SignatureValue)
```

Where the fields of ComponentOperationLabel are set to

component_id = ComponentID
label = Label

For signing operations, the ComponentID MUST be set to the ComponentID of the component performing the signature, and not to the ID of any other component. This means that a component cannot produce signatures in place of other component. However, components can verify signatures computed by other components. Domain separation is ensured by explicitly including the ComponentID with every operation.

4.4. Exported Secrets

The MLS Exporter functionality described in Section 8.5 of [RFC9420] does not provide forward security for exported secrets, because the exporter_secret is not deleted after a secret has been derived. In this section, we define a forward-secure exporter for use by application components.

The safe exporter is constructed from an Exporter Tree, tree of secrets with the same structure as the Secret Tree defined in Section 9 of [RFC9420], with two differences: First, an Exporter Tree always has 2^16 leaves, corresponding to the 16 bits of a ComponentID value. (As with the Secret Tree, the nodes of the tree can be generated on-demand, for space-efficiency.) Second, the root of the Exporter Tree is the application_export_secret, an additional secret derived from the epoch_secret at the beginning of the epoch in the same way as the other secrets listed in Table 4 of [RFC9420] using the label "application_export".

This tree defines one exported secret per ComponentID. The secret for a ComponentID is the tree_node_secret at the leaf node for that ComponentID:

SafeExportSecret(ComponentID) = tree_node_[LeafIndex(ComponentID)]_secret

Upon generating an exported secret for a component, the MLS implementation MUST regard that component’s exported secret as "consumed", and delete any source key material according to the deletion schedule in Section 9.2 of [RFC9420].

4.5. Pre-Shared Keys (PSKs)

PSKs represent key material that is injected into the MLS key schedule when creating or processing a commit as defined in Section 8.4 of [RFC9420]. Its injection into the key schedule means that all group members have to agree on the value of the PSK.

While PSKs are typically cryptographic keys which due to their properties add to the overall security of the group, the PSK mechanism can also be used to ensure that all members of a group agree on arbitrary pieces of data represented as octet strings (without the necessity of sending the data itself over the wire). For example, a component can use the PSK mechanism to enforce that all group members have access to and agree on a password or a shared file.

This is achieved by creating a new epoch via a PSK proposal. Transitioning to the new epoch requires using the information agreed upon.

To facilitate using PSKs safely, this document defines a new PSKType for application components. This provides domain separation between pre-shared keys used by the core MLS protocol and applications, and between those used by different components.

```
enum {
  // ...
  application(3),
  (255)
} PSKType;

struct {
  PSKType psktype;
  select (PreSharedKeyID.psktype) {
    // ...
    case application:
      ComponentID component_id;
      opaque psk_id<V>;
  };
  opaque psk_nonce<V>;
} PreSharedKeyID;
```

4.6. Attaching Application Data to MLS Messages

The MLS GroupContext, LeafNode, KeyPackage, and GroupInfo objects each have an extensions field that can carry additional data not defined by the MLS specification. The app_data_dictionary extension provides a generic container that applications can use to attach application data to these messages. Each usage of the extension serves a slightly different purpose:

* GroupContext: Confirms that all members of the group agree on the application data, and automatically distributes it to new joiners.

* KeyPackage and LeafNode: Associates the application data to a particular client, and advertises it to the other members of the group.

* GroupInfo: Distributes the application data confidentially to the new joiners for whom the GroupInfo is encrypted (as a Welcome message).

The content of the app_data_dictionary extension is a serialized AppDataDictionary object:

```
struct {
    ComponentID component_id;
    opaque data<V>;
} ComponentData;

struct {
    ComponentData component_data<V>;
} AppDataDictionary;
```

The entries in the component_data MUST be sorted by component_id, and there MUST be at most one entry for each component_id.

An app_data_dictionary extension in a LeafNode, KeyPackage, or GroupInfo can be set when the object is created. An app_data_dictionary extension in the GroupContext needs to be managed using the tools available to update GroupContext extensions. The creator of the group can set extensions unilaterally. Thereafter, the AppDataUpdate proposal described in the next section is used to update the app_data_dictionary extension.

4.7. Updating Application Data in the GroupContext

Updating the app_data_dictionary with a GroupContextExtensions proposal is cumbersome. The application data needs to be transmitted in its entirety, along with any other extensions, whether or not they are being changed. And a GroupContextExtensions proposal always requires an UpdatePath, which updating application state never should.

The AppDataUpdate proposal allows the app_data_dictionary extension to be updated without these costs. Instead of sending the whole value of the extension, it sends only an update, which is interpreted by the application to provide the new content for the app_data_dictionary extension. No other extensions are sent or updated, and no UpdatePath is required.

```
enum {
    invalid(0),
    update(1),
    remove(2),
    (255)
} AppDataUpdateOperation;

struct {
    ComponentID component_id;
    AppDataUpdateOperation op;

    select (AppDataUpdate.op) {
        case update: opaque update<V>;
        case remove: struct{};
    };
} AppDataUpdate;
```

An AppDataUpdate proposal is invalid if its component_id references a component that is not known to the application, or if it specifies the removal of state for a component_id that has no state present. A proposal list is invalid if it includes multiple AppDataUpdate proposals that remove state for the same component_id, or proposals that both update and remove state for the same component_id. In other words, for a given component_id, a proposal list is valid only if it contains (a) a single remove operation or (b) one or more update operation.

AppDataUpdate proposals are processed after any default proposals (i.e., those defined in [RFC9420]), and any AppEphemeral proposals (defined in Section 4.8).

When an MLS group contains the AppDataUpdate proposal type in the proposal_types list in the group’s required_capabilities extension, a GroupContextExtensions proposal MUST NOT add, remove, or modify the app_data_dictionary GroupContext extension. In other words, when every member of the group supports the AppDataUpdate proposal, a GroupContextExtensions proposal could be sent to update some other extension(s), but the app_data_dictionary GroupContext extension, if it exists, is left as it was.

A commit can contain a GroupContextExtensions proposal which modifies GroupContext extensions other than app_data_dictionary, and can be followed by zero or more AppDataUpdate proposals. This allows modifications to both the app_data_dictionary extension (via AppDataUpdate) and other extensions (via GroupContextExtensions) in the same Commit.

A client applies AppDataUpdate proposals by component ID. For each component_id field that appears in an AppDataUpdate proposal in the Commit, the client assembles a list of AppDataUpdate proposals with that component_id, in the order in which they appear in the Commit, and processes them in the following way:

* If the list comprises a single proposal with the op field set to remove:
  - If there is an entry in the component_states vector in the application_state extension with the specified component_id, remove it.
  - Otherwise, the proposal is invalid.
* If the list comprises one or more proposals, all with op field set to update:
  - Provide the application logic registered to the component_id value with the content of the update field from each proposal, in the order specified.
  - The application logic returns either an opaque value new_data that will be stored as the new application data for this component, or else an indication that it considers this update invalid.
  - If the application logic considers the update invalid, the MLS client MUST consider the proposal list invalid.
  - If no app_data_dictionary extension is present in the GroupContext, add one to the end of the extensions list in the GroupContext.
  - If there is an entry in the component_data vector in the app_data_dictionary extension with the specified component_id, then set its data field to the specified new_data.
  - Otherwise, insert a new entry in the component_states vector with the specified component_id and the data field set to the new_data value. The new entry is inserted at the proper point to keep the component_states vector sorted by component_id.
* Otherwise, the proposal list is invalid.

  NOTE: An alternative design here would be to have the update operation simply set the new value for the app_data_dictionary GCE, instead of sending a diff. This would be simpler in that the

MLS stack wouldn’t have to ask the application for the new state value, and would discourage applications from storing large state in the GroupContext directly (which bloats Welcome messages). It would effectively require the state in the GroupContext to be a hash of the real state, to avoid large AppDataUpdate proposals. This pushes some complexity onto the application, since the application has to define a hashing algorithm, and define its own scheme for initializing new joiners.

AppDataUpdate proposals do not require an UpdatePath. An AppDataUpdate proposal can be sent by an external sender. Likewise, AppDataUpdate proposals can be included in an external commit. Applications can make more restrictive validity rules for the update of their components, such that some components would not be valid at the application when sent in an external commit or via an external proposer.

4.8. Attaching Application Data to a Commit

The AppEphemeral proposal type allows an application component to associate application data to a Commit, so that the member processing the Commit knows that all other group members will be processing the same data. AppEphemeral proposals are ephemeral in the sense that they do not change any persistent state related to MLS, aside from their appearance in the transcript hash.

The content of an AppEphemeral proposal is the same as an app_data_dictionary extension. The proposal type is set in Section 7.

```
struct {
    ComponentID component_id;
    opaque data<V>;
} AppEphemeral;
```

An AppEphemeral proposal is invalid if it contains a component_id that is unknown to the application, or if the app_data_dictionary field contains any ComponentData entry whose data field is considered invalid by the application logic registered to the indicated component_id.

AppEphemeral proposals MUST be processed after any default proposals (i.e., those defined in [RFC9420]), but before any AppDataUpdate proposals.

A client applies an AppEphemeral proposal by providing the contents of the app_data_dictionary field to the component identified by the component_id. If a Commit references more than one AppEphemeral proposal for the same component_id value, then they MUST be processed in the order in which they are specified in the Commit.

AppEphemeral proposals do not require an UpdatePath. An AppEphemeral proposal can be sent by an external sender. Likewise, AppEphemeral proposals can be included in an external commit. Applications can make more restrictive validity rules for ephemeral updates of their components, such that some components would not be valid at the application when sent in an external commit or via an external proposer.

4.9. Safe Additional Authenticated Data (AAD)

Both MLS PublicMessage and PrivateMessage messages can contain arbitrary additional application-specific data. The corresponding authenticated_data field that conveys this application-specific data appears in the FramedContent struct for PublicMessage and directly in the PrivateMessage struct.

In an MLS PrivateMessage, the application_data is also present in intermediary structs: in the FramedContent, which is used to generate the message signature and tags, and in the PrivateContentAAD, which is used as the AAD input to the PrivateMessage.ciphertext.

The Safe AAD API defines a framing used to allow multiple application components to add AAD safely to the authenticated_data without conflicts or ambiguity.

When any AAD safe extension is included in the authenticated_data field, the "safe" AAD items MUST come before any non-safe data in the authenticated_data field. Safe AAD items are framed using the SafeAAD struct and are sorted in increasing numerical order of the component_id. The struct is described below:

```
struct {
  ComponentID component_id;
  opaque aad_item_data<V>;
} SafeAADItem;

struct {
  SafeAADItem aad_items<V>;
} SafeAAD;
```

If the SafeAAD is present or not in the authenticated_data is determined by the presence of the safe_aad component in the app_data_dictionary extension in the GroupContext (see Section 5). If safe_aad is present, but none of the "safe" AAD components have data to send in a particular message, the aad_items is a zero-length vector.

5. Negotiating Extensions and Components

MLS defines a Capabilities struct for LeafNodes (in turn used in KeyPackages), which describes which extensions are supported by the associated node.

However, that struct (defined in Section 7.2 of [RFC9420]) only has fields for a subset of the extensions possible in MLS, as reproduced below.

```
struct {
    ProtocolVersion versions<V>;
    CipherSuite cipher_suites<V>;
    ExtensionType extensions<V>;
    ProposalType proposals<V>;
    CredentialType credentials<V>;
} Capabilities;
```

The "MLS Extensions Types" registry represents extensibility of four core structs (GroupContext, GroupInfo, KeyPackage, and LeafNode) that have far reaching effects on the use of the protocol. The majority of MLS extensions in [RFC9420] extend one or more of these core structs.

Likewise, the required_capabilities GroupContext extension (defined in Section 11.1 of [RFC9420] and reproduced below) contains all mandatory to support non-default extensions in its extension_types vector. Its proposal_types vector contains any mandatory to support Proposals. Its credential_types vector contains any mandatory credential types.

```
struct {
   ExtensionType extension_types<V>;
   ProposalType proposal_types<V>;
   CredentialType credential_types<V>;
} RequiredCapabilities;
```

Due to an oversight in [RFC9420], the Capabilities struct does not include MLS Wire Formats. Instead, this document defines two extensions: supported_wire_formats (which can appear in LeafNodes), and required_wire_formats (which can appear in the GroupContext).

```
struct {
   WireFormat wire_formats<V>;
} WireFormats

WireFormats supported_wire_formats;
WireFormats requires_wire_formats;
```

This document also defines new components of the app_data_dictionary extension for supported and required Safe AAD, media types, and components.

The safe_aad component contains a list of components IDs. When present (in an app_data_dictionary extension) in a LeafNode, the semantic is the list of supported components that use Safe AAD. When present (in an app_data_dictionary extension) in the GroupContext, the semantic is the list of required Safe AAD components (those that must be understood by the entire group). If the safe_aad component is present, even with an empty list, (in the app_data_dictionary extension) in the GroupContext, then the authenticated_data field always starts with the SafeAAD struct defined in Section 4.9.

```
struct {
    ComponentID component_ids<V>;
} ComponentsList;

ComponentsList safe_aad;
```

The list of required and supported components follows the same model with the new component app_components. When present in a LeafNode, the semantic is the list of supported components. When present in the GroupContext, the semantic is the list of required components.

```
ComponentsList app_components;
```

Finally, the supported and required media types (formerly called MIME types) are communicated in the content_media_types component (see Section 6.2).

5.1. GREASE

The "Generate Random Extensions And Sustain Extensibility" (GREASE) approach is described in Section 13.5 of [RFC9420]. Further, Section 7.2 of [RFC9420] says that:

> "As discussed in Section 13, unknown values in capabilities MUST be ignored, and the creator of a capabilities field SHOULD include some random GREASE values to help ensure that other clients correctly ignore unknown values."

This specification adds the following locations where GREASE values for components can be included:

* LeafNode.capabilities.app_data_dictionary.safe_aad

* LeafNode.capabilities.app_data_dictionary.app_components

* LeafNode.extensions.app_data_dictionary

* KeyPackage.extensions.app_data_dictionary

* GroupInfo.extensions.app_data_dictionary

* app_ephemeral.app_data_dictionary

* authenticated_data.aad_items

Unknown values (including GREASE values) in any of these fields MUST be ignored by receivers. A random selection of GREASE values SHOULD be included in any of these fields that would otherwise be present.

6. Extensions

6.1. AppAck

An AppAck object is used to acknowledge receipt of application messages. Though this information implies no change to the group, it is conveyed inside an AppEphermeral Proposal with a component ID app_ack, so that it is included in the group's transcript by being included in Commit messages.

```
struct {
    uint32 sender;
    uint32 first_generation;
    uint32 last_generation;
} MessageRange;

struct {
    MessageRange received_ranges<V>;
} AppAck;
```

An AppAck represents a set of messages received by the sender in the current epoch. Messages are represented by the sender and generation values in the MLSCiphertext for the message. Each MessageRange represents receipt of a span of messages whose generation values form a continuous range from first_generation to last_generation, inclusive.

AppAck objects are sent as a guard against the Delivery Service dropping application messages. The sequential nature of the generation field provides a degree of loss detection, since gaps in the generation sequence indicate dropped messages. AppAck completes this story by addressing the scenario where the Delivery Service drops all messages after a certain point, so that a later generation is never observed. Obviously, there is a risk that AppAck messages could be suppressed as well, but their inclusion in the transcript means that if they are suppressed then the group cannot advance at all.

6.2. Content Advertisement

6.2.1. Description

This section defines a minimal framing format so MLS clients can signal which media type is being sent inside the MLS application_data object when multiple formats are permitted in the same group.

It also defines a new content_media_types application component which is used to indicate support for specific formats, using the extensive IANA Media Types registry (formerly called MIME Types). When the content_media_types component is present (in the app_data_dictionary extension) in a LeafNode, it indicates that node's support for a particular (non-empty) list of media types. When the content_media_types component is present (in the app_data_dictionary extension) in the GroupContext, it indicates a (non-empty) list of media types that need to be supported by all members of that MLS group, _and_ that the application_data will be framed using the application framing format described later in Section 6.2.4. This allows clients to confirm that all members of a group can communicate.

> Note that when the membership of a group changes, or when the policy of the group changes, it is responsibility of the committer to ensure that the membership and policies are compatible.

As clients are upgraded to support new formats they can use these extensions to detect when all members support a new or more efficient encoding, or select the relevant format or formats to send.

Vendor-specific media subtypes starting with vnd. can be registered with IANA without standards action as described in [RFC6838]. Implementations which wish to send multiple formats in a single application message, may be interested in the multipart/alternative media type defined in [RFC2046] or may use or define another type with similar semantics (for example using TLS Presentation Language syntax [RFC8446]).

Note that the usage of IANA media types in general does not imply the usage of MIME Headers [RFC2045] for framing.

6.2.2. Syntax

MediaType is a TLS encoding of a single IANA media type (including top-level type and subtype) and any of its parameters. Even if the parameter_value would have required formatting as a quoted-string in a text encoding, only the contents inside the quoted-string are included in parameter_value. Likewise, only the second character of a quoted-pair is included in parameter_value; the first escaping backslash ("") is omitted. MediaTypeList is an ordered list of MediaType objects.

```
struct {
    opaque parameter_name<V>;
    /* Note: parameter_value never includes the quotation marks of */
    /* an RFC 2045 quoted-string or the first "\" of a quoted-pair */
    opaque parameter_value<V>;
} Parameter;

struct {
    /* media_type is an IANA top-level media type, a "/" character,
     * and the IANA media subtype */
    opaque media_type<V>;

    /* a list of zero or more parameters defined for the subtype */
    Parameter parameters<V>;
} MediaType;

struct {
    /* must contain at least one item */
    MediaType media_types<V>;
} MediaTypeList;

MediaTypeList content_media_types;
```

Example IANA media types with optional parameters:

```
image/png
text/plain ;charset="UTF-8"
application/json
application/vnd.example.msgbus+cbor
```

For the example media type for text/plain, the media_type field would be text/plain, parameters would contain a single Parameter with a parameter_name of charset and a parameter_value of UTF-8.

6.2.3. Expected Behavior

An MLS client which implements this section SHOULD include the content_media_types component (in the app_data_dictionary extension) in its LeafNodes, listing all the media types it can receive. As usual, the client also includes content_media_types in the app_components list (in the app_data_dictionary extension) and support for the app_data_dictionary extension in its capabilities.extensions field in its LeafNodes (including in LeafNodes inside its KeyPackages).

When creating a new MLS group for an application using this specification, the group MAY include a content_media_types component (in the app_data_dictionary extension) in the GroupContext. (The creating client also includes its content_media_types component in its own LeafNode as described in the previous paragraph.)

MLS clients SHOULD NOT add an MLS client to an MLS group with content_media_types in its GroupContext unless the MLS client advertises it can support all the required MediaTypes. As an exception, a client could be preconfigured to know that certain clients support the required types. Likewise, an MLS client is already forbidden from issuing or committing a GroupContextExtensions Proposal which introduces required extensions which are not supported by all members in the resulting epoch.

6.2.4. Framing of application_data

When an MLS group contains the content_media_types component (in the app_data_dictionary extension) in its GroupContext, the application_data sent in that group is interpreted as ApplicationFraming as defined below:

```
struct {
    MediaType media_type;
    opaque<V> application_content;
} ApplicationFraming;
```

The media_type MAY be zero length, in which case, the media type of the application_content is interpreted as the first MediaType specified in the content_media_types component in the GroupContext.

6.3. SelfRemove Proposal

The design of the MLS protocol prevents a member of an MLS group from removing itself immediately from the group. (To cause an immediate change in the group, a member must send a Commit message. However, the sender of a Commit message knows the keying material of the new epoch and therefore needs to be part of the group.) Instead, a member wishing to remove itself can send a Remove Proposal and wait for another member to Commit its Proposal.

Unfortunately, MLS clients that join via an External Commit ignore pending, but otherwise valid, Remove Proposals. The member trying to remove itself has to monitor the group and send a new Remove Proposal in every new epoch until the member is removed. In a group with a burst of external joiners, a member connected over a high-latency link (or one that is merely unlucky) might have to wait several epochs to remove itself. A real-world situation in which this happens is a member trying to remove itself from a conference call as several dozen new participants are trying to join (often on the hour).

This section describes a new SelfRemove proposal type. It is designed to be included in External Commits.

6.3.1. Proposal Description

This document specifies a new MLS Proposal type called SelfRemove. Its syntax is described using the TLS Presentation Language [RFC8446] below (its content is an empty struct). It is allowed in External Commits and requires an UpdatePath. SelfRemove proposals are only allowed in a Commit by reference. SelfRemove cannot be sent as an external proposal.

```
struct {} SelfRemove;

struct {
    ProposalType msg_type;
    select (Proposal.msg_type) {
        case add:                       Add;
        case update:                    Update;
        case remove:                    Remove;
        case psk:                       PreSharedKey;
        case reinit:                    ReInit;
        case external_init:             ExternalInit;
        case group_context_extensions: GroupContextExtensions;
        case self_remove:               SelfRemove;
    };
} Proposal;
```

The description of behavior below only applies if all the members of a group support this proposal in their capabilities; such a group is a "self-remove-capable group".

An MLS client which supports this proposal can send a SelfRemove Proposal whenever it would like to remove itself from a self-remove-capable group. Because the point of a SelfRemove Proposal is to be available to external joiners (which are not yet members), these proposals MUST be sent in an MLS PublicMessage.

Whenever a member receives a SelfRemove Proposal, it includes it along with any other pending Propsals when sending a Commit. It already MUST send a Commit of pending Proposals before sending new application messages.

When a member receives a Commit referencing one or more SelfRemove Proposals, it treats the proposal like a Remove Proposal, except the leaf node to remove is determined by looking in the Sender leaf_index of the original Proposal. The member is able to verify that the Sender was a member.

Whenever a new joiner is about to join a self-remove-capable group with an External Commit, the new joiner MUST fetch any pending SelfRemove Proposals along with the GroupInfo object, and include the SelfRemove Proposals in its External Commit by reference. (An ExternalCommit can contain zero or more SelfRemove proposals). The new joiner MUST validate the SelfRemove Proposal before including it by reference, except that it skips the validation of the membership_tag because a non-member cannot verify membership.

During validation, SelfRemove proposals are processed after Update proposals and before Remove proposals. If there is a pending SelfRemove proposal for a specific leaf node and a pending Remove proposal for the same leaf node, the Remove proposal is invalid. A client MUST NOT issue more than one SelfRemove proposal per epoch.

The MLS Delivery Service (DS) needs to validate SelfRemove Proposals it receives (except that it cannot validate the membership_tag). If the DS provides a GroupInfo object to an external joiner, the DS SHOULD attach any SelfRemove proposals known to the DS to the GroupInfo object.

As with Remove proposals, clients need to be able to receive a Commit message which removes them from the group via a SelfRemove. If the DS does not forward a Commit to a removed client, it needs to inform the removed client out-of-band.

6.4. Last resort KeyPackages

Type: KeyPackage extension

6.4.1. Description

Section 10 of [RFC9420] details that clients are required to pre-publish KeyPackages so that other clients can add them to groups asynchronously. It also states that they should not be re-used:

> KeyPackages are intended to be used only once and SHOULD NOT be reused except in the case of a "last resort" KeyPackage (see Section 16.8). Clients MAY generate and publish multiple KeyPackages to support multiple cipher suites.

Section 16.8 of [RFC9420] then introduces the notion of last-resort KeyPackages as follows:

> An application MAY allow for reuse of a "last resort" KeyPackage in order to prevent denial-of-service attacks.

However, [RFC9420] does not specify how to distinguish regular KeyPackages from last-resort ones. The last_resort_key_package KeyPackage application component defined in this section fills this gap and allows clients to specifically mark KeyPackages as KeyPackages of last resort that MAY be used more than once in scenarios where all other KeyPackages have already been used.

The component allows clients that pre-publish KeyPackages to signal to the Delivery Service which KeyPackage(s) are meant to be used as last resort KeyPackages.

An additional benefit of using a component rather than communicating the information out-of-band is that the component is still present in Add proposals. Clients processing such Add proposals can authenticate that a KeyPackage is a last-resort KeyPackage and MAY make policy decisions based on that information.

6.4.2. Format

The purpose of the application component is simply to mark a given KeyPackage, which means it carries no additional data.

As a result, a LastResort Extension contains the component_id with an empty data field.

6.5. Multi-Credentials

Multi-credentials address use cases where there might not be a single credential that captures all of a client's authenticated attributes. For example, an enterprise messaging client may wish to provide attributes both from its messaging service, to prove that its user has a given handle in that service, and from its corporate owner, to prove that its user is an employee of the corporation. Multi-credentials can also be used in migration scenarios, where some clients in a group might wish to rely on a newer type of credential, but other clients haven't yet been upgraded.

New credential types MultiCredential and WeakMultiCredential are defined as shown below. These credential types are indicated with the values multi and weak-multi (see Section 7.4).

```
struct {
  CipherSuite cipher_suite;
  Credential credential;
  SignaturePublicKey credential_key;

  /* SignWithLabel(., "CredentialBindingTBS", CredentialBindingTBS) */
  opaque signature<V>;
} CredentialBinding

struct {
  CredentialBinding bindings<V>;
} MultiCredential;

struct {
  CredentialBinding bindings<V>;
} WeakMultiCredential;
```

The two types of credentials are processed in exactly the same way. The only difference is in how they are treated when evaluating support by other clients, as discussed below.

6.5.1. Credential Bindings

A multi-credential consists of a collection of "credential bindings". Each credential binding is a signed statement by the holder of the credential that the signature key in the LeafNode belongs to the holder of that credential. Specifically, the signature is computed using the MLS SignWithLabel function, with label "CredentialBindingTBS" and with a content that covers the contents of the CredentialBinding, plus the signature_key field from the LeafNode in which this credential will be embedded.

```
struct {
  CipherSuite cipher_suite;
  Credential credential;
  SignaturePublicKey credential_key;
  SignaturePublicKey signature_key;
} CredentialBindingTBS;
```

The cipher_suite for a credential is NOT REQUIRED to match the cipher suite for the MLS group in which it is used, but MUST meet the support requirements with regard to support by group members discussed below.

6.5.2. Verifying a Multi-Credential

A credential binding is supported by a client if the client supports the credential type and cipher suite of the binding. A credential binding is valid in the context of a given LeafNode if both of the following are true:

* The credential is valid according to the MLS Authentication Service.

* The credential_key corresponds to the specified credential, in the same way that the signature_key would have to correspond to the credential if the credential were presented in a LeafNode.

* The signature field is valid with respect to the signature_key value in the leaf node.

A client that receives a credential of type multi in a LeafNode MUST verify that all the following are true:

* All members of the group support credential type multi.

* For each credential binding in the multi-credential:

  - Every member of the group supports the cipher suite and credential type values for the binding.

  - The binding is valid in the context of the LeafNode.

A client that receives a credential of type weak-multi in a LeafNode MUST verify that all the following are true:

* All members of the group support credential type weak-multi.

* Each member of the group supports at least one binding in the multi-credential. (Different members may support different subsets.)

* Every binding that this client supports is valid in the context of the LeafNode.

7. IANA Considerations

This document requests the addition of various new values under the heading of "Messaging Layer Security". Each registration is organized under the relevant registry Type.

This document also requests the creation of a new MLS applications components registry as described in Section 7.5.

RFC EDITOR: Please replace XXXX throughout with the RFC number assigned to this document

7.1. MLS Wire Formats

7.2. MLS Extension Types

7.2.1. app_data_dictionary MLS Extension

The app_data_dictionary MLS Extension Type is used inside KeyPackage, LeafNode, GroupContext, or GroupInfo objects. It contains a sorted list of application component data objects (at most one per component).

* Value: 0x0006 (suggested)

* Name: app_data_dictionary

* Message(s): KP: This extension may appear in KeyPackage objects LN: This extension may appear in LeafNode objects GC: This extension may appear in GroupContext objects GI: This extension may appear in GroupInfo objects

* Recommended: Y

* Reference: RFC XXXX

7.2.2. supported_wire_formats MLS Extension

The supported_wire_formats MLS Extension Type is used inside LeafNode objects. It contains a list of non-default Wire Formats supported by the client node.

* Value: 0x0007 (suggested)

* Name: supported_wire_formats

* Message(s): LN: This extension may appear in LeafNode objects

* Recommended: Y

* Reference: RFC XXXX

7.2.3. required_wire_formats MLS Extension

The required_wire_formats MLS Extension Type is used inside GroupContext objects. It contains a list of non-default Wire Formats that are mandatory for all MLS members of the group to support.

* Value: 0x0008 (suggested)

* Name: required_wire_formats

* Message(s): GC: This extension may appear in GroupContext objects

* Recommended: Y

* Reference: RFC XXXX

7.3. MLS Proposal Types

7.3.1. AppDataUpdate Proposal

The app_data_update MLS Proposal Type is used to efficiently update application component data stored in the app_data_dictionary GroupContext extension.

* Value: 0x0008 (suggested)

* Name: app_data_update

* Recommended: Y

* External: Y

* Path Required: N

7.3.2. AppEphemeral Proposal

The app_ephemeral MLS Proposal Type is used to send opaque ephemeral application data that needs to be synchronized with a specific MLS epoch.

* Value: 0x0009 (suggested)
* Name: app_ephemeral
* Recommended: Y
* External: Y
* Path Required: N

7.3.3. SelfRemove Proposal

The self_remove MLS Proposal Type is used for a member to remove itself from a group more efficiently than using a remove proposal type, as the self_remove type is permitted in External Commits.

* Value: 0x000a (suggested)
* Name: self_remove
* Recommended: Y
* External: N
* Path Required: Y

7.3.4. AppAck Proposal

The app_ack MLS Proposal Type can be used by group members to acknowledge the receipt of application messages.

* Value: 0x000b (suggested)
* Name: app_ack
* Recommended: Y
* External: N
* Path Required: N

7.4. MLS Credential Types

7.4.1. Multi Credential

* Value: 0x0003 (suggested)
* Name: multi
* Recommended: Y
* Reference: RFC XXXX

7.4.2. Weak Multi Credential

* Value: 0x0004
* Name: weak-multi
* Recommended: Y
* Reference: RFC XXXX

7.4.3. CredentialBindingTBS

* Label: "CredentialBindingTBS"
* Recommended: Y
* Reference: RFC XXXX

7.5. MLS Component Types

This document requests the creation of a new IANA "MLS Component Types" registry under the "Messaging Layer Security" group registry heading. Assignments to this registry in the range 0x0000 0000 to 0x7FFF FFFF are via Specification Required policy [RFC8126] using the MLS Designated Experts. Assignments in the range 0x8000 0000 to 0xFFFF FFFF are for private use.

Template:

* Value: The numeric value of the component ID
* Name: The name of the component
* Where: The objects(s) in which the component may appear, drawn from the following list:

- AD: SafeAAD objects
- AE: AppEpheral proposals
- ES: Exporter Secret labels
- GC: GroupContext objects
- GI: GroupInfo objects
- HP: HPKE key labels
- KP: KeyPackage objects
- LN: LeafNode objects
- PS: PSK labels
- SK: Signature Key labels

* Recommended: Same as in Section 17.1 of [RFC9420]

* Reference: The document where this component is defined

The restrictions noted in the "Where" column are to be enforced by the application. MLS implementations MUST NOT impose restrictions on where component IDs are used in which parts of MLS, unless specifically directed to by the application.

Initial Contents:

Value	Name	Where	R	Ref
0x0000 0000	RESERVED	N/A	-	RFCXXXX
0x0000 0001	app_components	LN,GC	Y	RFCXXXX
0x0000 0002	safe_aad	LN,GC	Y	RFCXXXX
0x0000 0003	content_media_types	LN,GC	Y	RFCXXXX
0x0000 0004	last_resort_key_package	KP	Y	RFCXXXX
0x0000 0005	app_ack	AE	Y	RFCXXXX
0x0000 0a0a	GREASE	Note1	Y	RFCXXXX

0x0000 1a1a	GREASE	Note1	Y	RFCXXXX
0x0000 2a2a	GREASE	Note1	Y	RFCXXXX
0x0000 3a3a	GREASE	Note1	Y	RFCXXXX
0x0000 4a4a	GREASE	Note1	Y	RFCXXXX
0x0000 5a5a	GREASE	Note1	Y	RFCXXXX
0x0000 6a6a	GREASE	Note1	Y	RFCXXXX
0x0000 7a7a	GREASE	Note1	Y	RFCXXXX
0x0000 8a8a	GREASE	Note1	Y	RFCXXXX
0x0000 9a9a	GREASE	Note1	Y	RFCXXXX
0x0000 aaaa	GREASE	Note1	Y	RFCXXXX
0x0000 baba	GREASE	Note1	Y	RFCXXXX
0x0000 caca	GREASE	Note1	Y	RFCXXXX
0x0000 dada	GREASE	Note1	Y	RFCXXXX
0x0000 eaea	GREASE	Note1	Y	RFCXXXX
0x8000 0000 -				
0xFFFF FFFF	Reserved for Private Use	N/A	N	RFCXXXX

Table 1

Note1: GREASE values for components MAY be present in AD, AE, GI, KP, and LN objects.

8. Security considerations

8.1. Safe Application API

The Safe Application API provides the following security guarantee: If an application uses MLS with application components, the security guarantees of the base MLS protocol and the security guarantees of each application component analyzed in isolation, still hold for the composed application of the MLS protocol. In other words, the Safe Application API protects applications from careless component

developers. It is not possible that a combination of components (the developers of which did not know about each other) impedes the security of the base MLS protocol or any other component. No further analysis of the combination is necessary. This also means that any security vulnerabilities introduced by one component do not spread to other components or the base MLS implementation.

8.2. AppAck

When AppAck objects are received, they allow clients to detect if the Delivery Service (or an intermediary) dropped application messages, since gaps in the generation sequence indicate dropped messages. When AppAck messages are accepted by the Delivery Service, but not received by some members, the members who have missed the corresponding AppEphemeral proposals will not be able to send or receive a commit message, because the proposal is included in the transcript hash. Likewise, if AppAck objects and/or commits are sent periodically by every member, other members will be able to detect a member that is no longer sending on that schedule or whose handshake messages are being suppressed by the DS.

> Note: External Commits do not typically contain pending proposals (including AppEphemeral proposals). Client that send an AppAck component in an AppEphemeral proposal will need to send a new AppAck component in an AppEphemeral proposal (in the new epoch) after receiving an External Commit until it has been incorporated into an accepted Commit.

The schedule on which AppAck objects are sent in AppEphemeral proposals is up to the application, and determines which cases of loss/suppression are detected. For example:

* The application might have the committer include an AppAck whenever a Commit is sent, so that other members could know when one of their messages did not reach the committer.

* The application could have a client send an AppAck whenever an application message is sent, covering all messages received since its last AppAck. This would provide a complete view of any losses experienced by active members.

* The application could simply have clients send AppAck proposals on a timer, so that all participants' state would be known.

An application using AppAck to guard against loss/suppression of application messages also needs to ensure that AppAck messages and the Commits that reference them are not dropped. One way to do this is to always encrypt Proposal and Commit messages, to make it more

difficult for the Delivery Service to recognize which messages contain AppAcks. The application can also have clients enforce an AppAck schedule, reporting loss if an AppAck is not received at the expected time.

8.3. Content Advertisement

Use of the content_media_types component could leak some private information visible in KeyPackages and inside an MLS group. This could be used to infer a specific implementation, platform, or even version. Clients should carefully consider the privacy implications in their environment of making a list of acceptable media types available.

Implementations need to be prepared to parse media types containing long parameter lists, potentially containing characters which would be escaped or quoted in [RFC5322].

8.4. SelfRemove

An external recipient of a SelfRemove Proposal cannot verify the membership_tag. However, an external joiner also has no way to completely validate a GroupInfo object that it receives. An insider can prevent an External Join by providing either an invalid GroupInfo object or an invalid SelfRemove Proposal. The security properties of external joins does not change with the addition of this proposal type.

8.5. Multi Credentials

Using a Weak Multi Credential reduces the overall credential security to the security of the least secure of its credential bindings.

9. References

9.1. Normative References

[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/rfc/rfc2119>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/rfc/rfc5322>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[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>.

9.2. Informative References

[I-D.ietf-mls-architecture] Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., and A. Duric, "The Messaging Layer Security (MLS) Architecture", Work in Progress, Internet-Draft, draft-ietf-mls-architecture-15, 3 August 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-15>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/rfc/rfc2045>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/rfc/rfc2046>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

Appendix A. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-07 - add AppAck to IANA considerations - adapt Safe AAD scope - remove targeted messages altogether with intention to publish it as a separate document - assign self_remove proposal to a non duplicate number (0x000a) - refactor TargetedMessage to no longer use structs removed from when it was a "safe extension" - remove the no-longer needed targeted_messages_capability (now signaled using support_wire_formats and required_wire_formats) - add TargetedMessageTBS and CredentialBundleTBS to MLS Signature Labels IANA registry - add GREASE values for components - fix safe exporter definition - resolve TODOs from -06 - fix numerous typos

draft-06

* Integrate notion of Application API from draft-barnes-mls-appsync

draft-05

* Include definition of ExtensionState extension

* Add safe use of AAD to Safe Extensions framework

* Clarify how capabilities negotiation works in Safe Extensions framework

draft-04

* No changes (prevent expiration)

draft-03

* Add Last Resort KeyPackage extension

* Add Safe Extensions framework

* Add SelfRemove Proposal

draft-02

* No changes (prevent expiration)

draft-01

* Add Content Advertisement extensions

draft-00

* Initial adoption of draft-robert-mls-protocol-00 as a WG item.

* Add Targeted Messages extension (*)

Contributors

Joel Alwen
Amazon
Email: alwenjo@amazon.com

Konrad Kohbrok
Phoenix R&D
Email: konrad.kohbrok@datashrine.de

Rohan Mahy
Rohan Mahy Consulting Services
Email: rohan.ietf@gmail.com

Marta Mularczyk
Amazon
Email: mulmarta@amazon.com

Richard Barnes
Cisco Systems
Email: rlb@ipv.sx

Author’s Address

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