Messaging Layer Security
Internet-Draft
Intended status: Informational
Expires: 5 September 2024

R. Barnes
S. Nandakumar
Cisco

4 March 2024

Additional MLS Credentials
draft-barnes-mls-addl-creds-01

Abstract

This specification defines two new kinds of credentials for use within the Message Layer Security (MLS) credential framework: UserInfo Verifiable Credentials and multi-credentials. UserInfo Verifiable Credentials allow clients to present credentials that associate OpenID Connect attributes to a signature key pair held by the client. Multi-credentials allow clients to present authenticated attributes from multiple sources, or to present credentials in different formats to support groups with heterogeneous credential support.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-barnes-mls-addl-creds/.

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/bifurcation/mls-userinfo-vc.

Status of This Memo

Table of Contents

1. Introduction

MLS provides end-to-end authenticated key exchange [I-D.ietf-mls-protocol]. Each client participating in an MLS group is authenticated with a credential. The MLS credential structure is extensible: New MLS credential formats can be defined which support new mechanisms for authenticating clients.

In this document, we define two new types of credential:

* Credentials based on OpenID Connect UserInfo Verifiable Credentials

* Multi-credentials that present several credentials at once

UserInfo Verifiable Credentials (VCs) are a mechanism by which an OpenID Provider can bind user attributes to a signature key pair. OpenID Connect is already widely deployed as a mechanism for connecting authentication services to applications, and the OpenID Foundation is in the process of standardizing the extensions required for OpenID Providers to issue UserInfo VCs.

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.

2. Terminology

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 RFC 2119 [RFC2119].

This specification uses terms from the MLS Protocol specification. In particular, we refer to the MLS Credential object, which represents an association between a client's identity and the signature key pair that the client will use to sign messages in the MLS key exchange protocol.

3. UserInfo Verifiable Credentials

As described in the MLS architecture, MLS requires an Authentication Service (AS) as well as a Delivery Service (DS) [I-D.ietf-mls-architecture]. The full security goals of MLS are only realized if the AS and DS are non-colluding. In other words, applications can deploy MLS to get end-to-end encryption (acting as MLS Delivery Service), but they need to partner with a non-colluding Authentication Service in order to achieve full end-to-end security.

OpenID Connect is widely used to integrate identity providers with applications, but its current core protocol doesn't provide the binding to cryptographic keys required for use in MLS. When OpenID Connect is coupled with the "Verifiable Credentials" framework, however, it can be used to provision clients with signed "UserInfo VC" objects that contain the critical elements of a credential to be used in MLS:

* Identity attributes for the user of a client

* A public key whose private key is held by a client

* A signature over the above by a trusted identity provider

The required updates to OpenID Connect are specfied in [OpenIDUserInfoVC]. That document defines a profile of the OpenID for Verifiable Credential Issuance protocol for issuing "UserInfo Verifiable Credentials". These credentials bind a signature key pair to the user attributes typically exposed through the OpenID Connect UserInfo endpoint.

A "UserInfoVC" credential encapsulates a UserInfo Verifiable Credential object, so that it can be used for authenticating an MLS client. We also describe the validation process that MLS clients use to verify UserInfoVC objects that they receive via MLS.

3.1. UserInfo VC Life-Cycle

```
      +----+
      |    | (1) Generate signature key pair
      |    V
   +----------+                                   +----------+
   |          |<~~~(2) OpenID Connect Login~~~~~~>|          |
   |          |                                   |          |
   |          |                                   |          |
   |          |-------(3) Credential Request----->|  OpenID  |
   | Client 1 |     (type=UserInfoCredential,     | Provider |
   |          |      token & proof)               |   (OP)   |
   |          |                                   |          |
   |          |<------(4) Credential Response-----|          |
   |          |         (credential)              |          |
   +----------+                                   +----------+
         :                                             ^
         : (5) UserInfoVC in MLS KeyPackage            |
         :                                             |
         v                                             |
   +----------+                                        |
   |          |                                        |
   |          | (6) Fetch JWK set, Verify JWT          |
   |          |        Signature                       |
   | Client 2 |<----------------------------------------
   |          |----+
   |          |    | (7) Validate vc claim using
   |          |<---+     OP's JWK
   +----------+
```

               OpenID Connect UserInfo VC MLS Credential Flow

Figure 1: The protocol interactions to issue and verify a UserInfo VC

The basic steps showing OIDC Verifiable Credential based MLS credential flow are shown in Figure 1.

Client 1 is an MLS client that acts as a Holder in the VC model. Client 2 is also an MLS client, but acts in the Verifier role in the VC model. Both clients implement certain OpenID Connect operations to obtain and verify UserInfo VC objects.

1. Client 1 generates a signature key pair using an algorithm that is supported by both MLS and UserInfo VC.

2. Client 1 performs an OpenID Connect login interaction with the scope "userinfo_credential" to obtain UserInfo VCs.

3. Client 1 sends a Credential Request specifying that it desires a UserInfo VC, together with a proof that it controls the private key of a signature key pair and the access token.

4. The OpenID Provider verifies the proof and create a Credential Response containing the UserInfo VC attesting the claims that would have been provided by the UserInfo endpoint and public key corresponding to the private key used to compute the proof in the Credential Request.

5. Client 1 generates a UserInfoVC MLS Credential object with the signed UserInfo VC JWT. Client 1 embeds the UserInfoVC in an MLS KeyPackage object and signs the KeyPackage object with the corresponding private key.

6. Client 1 sends the KeyPackage to Client 2, e.g., by posting it to a directory from which Client 2 fetches it when it wants to add Client 1 to a group.

7. Client 2 verifies the signature on the KeyPackage and extracts the UserInfoVC credential. Client 2 uses OpenID Connect Discovery to fetch the OpenID Provider’s JWK set.

8. Client 2 verifies the signed UserInfo VC using the the appropriate key from the OpenID Provider’s JWK set.

If all checks pass, Client 2 has a high degree of assurance of the identity of Client 1. At this point Client 1’s KeyPackage (including the VerifiableCredential) will be included in the MLS group’s ratchet tree and distributed to the other members of the group. The other members of the group can verify the VerifiableCredential in the same way as Client 2.

3.2. UserInfoVC

A new credential type UserInfoVC is defined as shown below. This credential type is indicated with the CredentialType value userinfo_vc (see Section 7).

```
struct {
    opaque jwt<0..2^32-1>;
} UserInfoVC;
```

The jwt field contains the signed JWT-formatted UserInfo VC object (as defined in [OpenIDUserInfoVC]), encoded using UTF-8. The payload of object MUST provide iss and vc claims. The iss claim is used to look up the OpenID Provider's metadata. The vc claim contains authenticated user attributes and a public key binding. Specifically, the field vc.credentialSubject.id contains a did:jwk URI describing the subject's public key as a JWK.

3.3. Credential Validation

An MLS client validates a UserInfoVC credential in the context of an MLS LeafNode with the following steps:

* Verify that the jwt field parses successfully into a JWT [!@RFC7519], whose payload parses into UserInfo object as defined in Section 5.3.2 of [!@OpenID].

* Verify that an iss claim is present in the UserInfo VC payload and that "iss" value represents and issuer that is trusted according to the client's local policy.

* Verify the JWT signature:

  - Fetch the issuer metadata using OIDC Discovery [@!OpenID.Discovery].

  - Use the jwks_uri field in the metadata to fetch the JWK set.

  - Verify that the JWT signature verifies under one of the keys in the JWK set.

* Verify the key binding:

  - Verify that a vc claim is present in the UserInfo VC payload.

  - Verify that the value of the claim is a JSON object that contains a credentialSubject field, as defined in Section 4 of openid-userinfo-vc.

  - Verify id field exists and it MUST be a a Decentralized Identifier with DID method jwk (W3c.did-core).

  - Verify that the jwk field parses as a JWK.

  - Verify that the signature_key in the LeafNode matches the key in the id field.

If all of the above checks pass, the client can use the signature key in the JWK for verifying MLS signatures using the signature scheme corresponding to the kty and crv parameters in the JWK. The identity attributes in the JWT should be associated with the MLS client that presented the credential.

3.4. Mapping between JWK Key Types and MLS Ciphersuites

Below table maps JWK key types (kty) and elliptic curves (crv) to the equivalent MLS signature scheme.

kty	crv	TLS/MLS signature scheme
EC	P-256	ECDSA with P-256 and SHA-256
EC	P-384	ECDSA with P-384 and SHA-384
EC	P-521	ECDSA with P-521 and SHA-512
EC	Ed25519	Ed25519
EC	Ed448	Ed448

Table 1

4. Multi-Credentials

New credential types MultiCredential and WeakMultiCredential are defined as shown below. These credential types are indicated with CredentialType values multi and weak-multi (see Section 7).

```
   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.

4.1. 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.

4.2. 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 of 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 of the following are true:

* All members of the group support credential type 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.

5. Security Considerations

The validation procedures for UserInfoVC credentials verify that a JWT came from a given issuer. It doesn’t verify that the issuer is authorative for the claimed attributes. The client needs to verify that the issuer is trusted to assert the claimed attributes.

6. Privacy Considerations

UserInfo can contain sensitive info such as human names, phone numbers, and using these credentials in MLS will expose this information to other group members, and potentially others if used in a prepublished KeyPackage.

7. IANA Considerations

7.1. MLS Credential Types

IANA is requested to register add the following new entries to the MLS Credential Type registry.

Value	Name	Recommended	Reference
0x0003	userinfo-vc	Y	RFC XXXX
0x0004	multi	Y	RFC XXXX
0x0005	weak-multi	Y	RFC XXXX

Table 2

8. Normative 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-12, 3 March 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-12>.

[I-D.ietf-mls-protocol]
Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", Work in Progress, Internet-Draft, draft-ietf-mls-protocol-20, 27 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-20>.

[OpenIDUserInfoVC]
Ansari, M., Barnes, R., Kasselman, P., and K. Yasuda, "OpenID Connect UserInfo Verifiable Credentials 1.0", 15 December 2022, <https://openid.net/specs/openid-connect-userinfo-vc-1_0.html>.

Authors' Addresses

Richard Barnes
Cisco
Email: rlb@ipv.sx

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 25 April 2024

R. Robert
Phoenix R&D

23 October 2023

The Messaging Layer Security (MLS) Extensions
draft-ietf-mls-extensions-03

Abstract

This document describes extensions to the Messaging Layer Security (MLS) protocol.

Discussion Venues

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

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 describes extensions to [mls-protocol] that are not part of the main protocol specification. The protocol specification includes a set of core extensions that are likely to be useful to many applications. The extensions described in this document are intended to be used by applications that need to extend the MLS protocol.

1.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

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 (*)

2. Safe Extensions

The MLS specification is extensible in a variety of ways (see Section 13 of the [RFC9420]) and describes the negotiation and other handling of extensions and their data within the protocol. However, it does not provide guidance on how extensions can or should safely interact with the base MLS protocol. The goal of this section is to simplify the task of developing MLS extensions.

More concretely, this section defines the Safe Extension API, a library of extension components which simplifies development and security analysis of extensions, provides general guidance on using the built-in functionality of the base MLS protocol to build extensions, defines specific examples of extensions built on top of the Safe Extension API alongside the built-in mechanisms of the base MLS protocol, defines a number of labels registered in IANA which can be safely used by extensions, so that the only value an extension developer must add to the IANA registry themselves is a unique ExtensionType.

2.1. Safe Extension API

The Safe Extension API is a library that defines a number of components from which extensions can be built. In particular, these components provide extensions the ability to:

* Make use of selected private and public key material from the MLS specification, e.g. to encrypt, decrypt, sign, verify and derive fresh key material.

* Inject key material via PSKs in a safe way to facilitate state agreement without the use of a group context extension.

* Export secrets from MLS in a way that, in contrast to the built-in export functionality of MLS, preserves forward secrecy of the exported secrets within an epoch.

The Safe Extension API is not an extension itself, it only defines components from which other extensions can be built. Some of these components modify the MLS protocol and, therefore, so do the extensions built from them.

Where possible, the API makes use of mechanisms defined in the MLS specification. For example, part of the safe API is the use of the SignWithLabel function described in Section 5.1.2 of [RFC9420].

2.1.1. Security

An extension is called safe if it does not modify the base MLS protocol or other MLS extensions beyond using components of the Safe Extension API. The Safe Extension API provides the following security guarantee: If an application uses MLS and only safe MLS extensions, then the security guarantees of the base MLS protocol and the security guarantees of safe extensions, each analyzed in isolation, still hold for the composed extended MLS protocol. In other words, the Safe Extension API protects applications from careless extension developers. As long as all used extensions are safe, it is not possible that a combination of extensions (the developers of which did not know about each other) impedes the security of the base MLS protocol or any used extension. No further analysis of the combination is necessary. This also means that any security vulnerabilities introduced by one extension do not spread to other extensions or the base MLS.

2.1.2. Common Data Structures

Most components of the Safe Extension API use the value ExtensionType which is a unique uint16 identifier assigned to an extension in the MLS Extension Types IANA registry (see Section 17.3 of [RFC9420]).

Most Safe Extension API components also use the following data structure, which provides domain separation by extension_type of various extension_data.

```
struct {
  ExtensionType extension_type;
  opaque extension_data<V>;
} ExtensionContent;
```

Where extension_type is set to the type of the extension to which the extension_data belongs.

If in addition a label is required, the following data structure is used.

```
struct {
  opaque label;
  ExtensionContent extension_content;
} LabeledExtensionContent;
```

2.1.3. Hybrid Public Key Encryption (HPKE)

This component of the Safe Extension API allows extensions to make use of all HPKE key pairs generated by MLS. An extension identified by an ExtensionType 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 LabeledExtensionContent with extension_type set to ExtensionType. The extension_data can be set to an arbitrary Context specified by the extension designer (and can be empty if not needed). For example, an extension can use a key pair PublicKey, PrivateKey to encrypt data as follows:

```
   SafeEncryptWithContext(ExtensionType, PublicKey, Context, Plaintext) =
       SealBase(PublicKey, LabeledExtensionContent, "", Plaintext)

   SafeDecryptWithContext(ExtensionType, PrivateKey, Context, KEMOutput, Cipherte
xt) =
       OpenBase(KEMOutput, PrivateKey, LabeledExtensionContent, "", Ciphertext)
```

Where the fields of LabeledExtensionContent are set to

```
   label = "MLS 1.0 ExtensionData"
   extension_type = ExtensionType
   extension_data = Context
```

For operations involving the secret key, ExtensionType MUST be set to the ExtensionType of the implemented extension, and not to the type of any other extension. In particular, this means that an extension cannot decrypt data meant for another extension, while extensions can encrypt data to other extensions.

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 extension). 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 nodes 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.

2.1.4. Signature Keys

MLS session states contain a number of signature keys including the ones in the LeafNode structs. Extensions 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, an extension identified by ExtensionType should sign and verify using:

```
   SafeSignWithLabel(ExtensionType, SignatureKey, Label, Content) =
       SignWithLabel(SignatureKey, "LabeledExtensionContent", LabeledExtensionCon
tent)

   SafeVerifyWithLabel(ExtensionType, VerificationKey, Label, Content, SignatureV
alue) =
       VerifyWithLabel(VerificationKey, "LabeledExtensionContent", LabeledExtensi
onContent, SignatureValue)
```

Where the fields of LabeledExtensionContent are set to

```
   label = Label
   extension_type = ExtensionType
   extension_data = Content
```

For signing operations, the ExtensionType MUST be set to the ExtensionType of the implemented extension, and not to the type of any other extension. In particular, this means that an extension cannot produce signatures in place of other extensions. However, extensions can verify signatures computed by other extensions. Note that domain separation is ensured by explicitly including the ExtensionType with every operation.

2.1.5. Exporting Secrets

An extension can use MLS as a group key agreement protocol by exporting symmetric keys. Such keys can be exported (i.e. derived from MLS key material) in two phases per epoch: Either at the start of the epoch, or during the epoch. Derivation at the start of the epoch has the added advantage that the source key material is deleted after use, allowing the derived key material to be deleted later even during the same MLS epoch to achieve forward secrecy. The following protocol secrets can be used to derive key from for use by extensions:

* epoch_secret at the beginning of an epoch

* extension_secret during an epoch

The extension_secret is 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 "extension".

Any derivation performed by an extension either from the epoch_secret or the extension_secret has to use the following function:

```
DeriveExtensionSecret(Secret, Label) =
  ExpandWithLabel(Secret, "ExtensionExport " + ExtensionType + " " + Label)
```

Where ExpandWithLabel is defined in Section 8 of [RFC9420] and where ExtensionType MUST be set to the ExtensionType of the implemented extension.

2.1.6. 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, an extension 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 in a safe way, this document defines a new PSKType for extensions. This provides domain separation between pre-shared keys used by the core MLS protocol and applications, and between those used by different extensions.

```
   enum {
     reserved(0),
     external(1),
     resumption(2),
     extensions(3),
     (255)
   } PSKType;

   struct {
     PSKType psktype;
     select (PreSharedKeyID.psktype) {
       case external:
         opaque psk_id<V>;

       case resumption:
         ResumptionPSKUsage usage;
         opaque psk_group_id<V>;
         uint64 psk_epoch;

       case extensions:
         ExtensionType extension_type;
         opaque psk_id<V>;
     };
     opaque psk_nonce<V>;
   } PreSharedKeyID;
```

2.1.7. Extension Designer Tools

The safe extension API allows extension designers to sign and encrypt payloads without the need to register their own IANA labels. Following the same pattern, this document also provides ways for extension designers to define their own wire formats, proposals and credentials.

2.1.7.1. Wire Formats

Extensions can define their own MLS messages by using the mls_extension_message MLS Wire Format. The mls_extension_message Wire Format is IANA registered specifically for this purpose and extends the select statement in the MLSMessage struct as follows:

```
   case mls_extension_message:
       ExtensionContent extension_content;
```

The extension_type in extension_content MUST be set to the type of the extension in question. Processing of self-defined wire formats has to be defined by the extension.

2.1.7.2. Proposals

Similar to wire formats, extensions can define their own proposals by using one of three dedicated extension proposal types: extension_proposal, extension_path_proposal and extension_external_propsal. Each type contains the same ExtensionContent struct, but is validated differently: extension_proposal requires no UpdatePath and can not be sent by an external sender extension_path_proposal requires an UpdatePath and can not be sent by an external sender extensions_external_proposal requires no UpdatePath and can be sent by an external sender.

Each of the three proposal types is IANA registered and extends the select statement in the Proposal struct as follows:

```
case extension_proposal:
    ExtensionContent extension_content;
case extension_path_proposal:
    ExtensionContent extension_content;
case extension_external_proposal:
    ExtensionContent extension_content;
```

The extension_type MUST be set to the type of the extension in question.

Processing and validation of self-defined proposals has to be defined by the extension. However, validation rules can lead to a previously valid commit to become invalid, not the other way around. This is with the exception of proposal validation for external commits, where self-defined proposals can be declared valid for use in external commits. More concretely, if an external commit is invalid, only because the self-defined proposal is part of it (the last rule in external commit proposal validation in Section 12.2 of [RFC9420]), then the self-defined validation rules may rule that the commit is instead valid.

2.1.7.3. Credentials

Extension designers can also define their own credential types via the IANA registered extension_credential credential type. The extension_credential extends the select statement in the Credential struct as follows:

```
case extension_credential:
    ExtensionContent extension_content;
```

The extension_type in the extension_content must be set to that of the extension in question with the extension_data containing all other relevant data. Note that any credential defined in this way has to meet the requirements detailed in Section 5.3 of the MLS specification.

2.2. Extension Design Guidance

While extensions can modify the protocol flow of MLS and the associated properties in arbitrary ways, the base MLS protocol already enables a number of functionalities that extensions can use without modifying MLS itself. Extension authors should consider using these built-in mechanisms before employing more intrusive changes to the protocol.

2.2.1. Storing State in Extensions

Every type of MLS extension can have data associated with it and, depending on the type of extension (KeyPackage Extension, GroupContext Extension, etc.) that data is included in the corresponding MLS struct. This allows the authors of an extension to make use of any authentication or confidentiality properties that the struct is subject to as part of the protocol flow.

* GroupContext Extensions: Any data in a group context extension is agreed-upon by all members of the group in the same way as the rest of the group state. As part of the GroupContext, it is also sent encrypted to new joiners via Welcome messages and (depending on the architecture of the application) may be available to external joiners. Note that in some scenarios, the GroupContext may also be visible to components that implement the delivery service.

* GroupInfo Extensions: GroupInfo extensions are included in the GroupInfo struct and thus sent encrypted and authenticated by the signer of the GroupInfo to new joiners as part of Welcome messages. It can thus be used as a confidential and authenticated channel from the inviting group member to new joiners. Just like GroupContext extensions, they may also be visible to external joiners or even parts of the delivery service. Unlike GroupContext extensions, the GroupInfo struct is not part of the group state that all group members agree on.

* KeyPackage Extensions: KeyPackages (and the extensions they include) are pre-published by individual clients for asynchronous group joining. They are included in Add proposals and become part of the group state once the Add proposal is committed. They are, however, removed from the group state when the owner of the

KeyPackage does the first commit with a path. As such, KeyPackage extensions can be used to communicate data to anyone who wants to invite the owner to a group, as well as the other members of the group the owner is added to. Note that KeyPackage extensions are visible to the server that provides the KeyPackages for download, as well as any part of the delivery service that can see the public group state.

* LeafNode Extensions: LeafNodes are a part of every KeyPackage and thus follow the same lifecycle. However, they are also part of any commit that includes an UpdatePath and clients generally have a leaf node in each group they are a member of. Leaf node extensions can thus be used to include member-specific data in a group state that can be updated by the owner at any time.

3. Extensions

3.1. AppAck

Type: Proposal

3.1.1. Description

An AppAck proposal is used to acknowledge receipt of application messages. Though this information implies no change to the group, it is structured as a Proposal message 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 proposal 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 proposals 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.

The schedule on which sending AppAck proposals are sent 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 proposal 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 proposals 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.

3.2. Targeted messages

3.2.1. Description

MLS application messages make sending encrypted messages to all group members easy and efficient. Sometimes application protocols mandate that messages are only sent to specific group members, either for privacy or for efficiency reasons.

Targeted messages are a way to achieve this without having to create a new group with the sender and the specific recipients which might not be possible or desired. Instead, targeted messages define the format and encryption of a message that is sent from a member of an existing group to another member of that group.

The goal is to provide a one-shot messaging mechanism that provides confidentiality and authentication.

Targeted Messages makes use the Safe Extension API as defined in Section 2.1. reuse mechanisms from [mls-protocol], in particular [hpke].

3.2.2. Format

This extension uses the mls_extension_message WireFormat as defined in Section Section 2.1.7.1, where the content is a TargetedMessage.

```
struct {
  opaque group_id<V>;
  uint64 epoch;
  uint32 recipient_leaf_index;
  opaque authenticated_data<V>;
  opaque encrypted_sender_auth_data<V>;
  opaque hpke_ciphertext<V>;
} TargetedMessage;

enum {
  hpke_auth_psk(0),
  signature_hpke_psk(1),
} TargetedMessageAuthScheme;

struct {
  uint32 sender_leaf_index;
  TargetedMessageAuthScheme authentication_scheme;
  select (authentication_scheme) {
    case HPKEAuthPsk:
    case SignatureHPKEPsk:
      opaque signature<V>;
  }
  opaque kem_output<V>;
} TargetedMessageSenderAuthData;

struct {
  opaque group_id<V>;
  uint64 epoch;
  uint32 recipient_leaf_index;
  opaque authenticated_data<V>;
  TargetedMessageSenderAuthData sender_auth_data;
} TargetedMessageTBM;

struct {
  opaque group_id<V>;
  uint64 epoch;
```

```
     uint32 recipient_leaf_index;
     opaque authenticated_data<V>;
     uint32 sender_leaf_index;
     TargetedMessageAuthScheme authentication_scheme;
     opaque kem_output<V>;
     opaque hpke_ciphertext<V>;
   } TargetedMessageTBS;

   struct {
     opaque group_id<V>;
     uint64 epoch;
     opaque label<V> = "MLS 1.0 targeted message psk";
   } PSKId;
```

Note that TargetedMessageTBS is only used with the TargetedMessageAuthScheme.SignatureHPKEPsk authentication mode.

3.2.3. Encryption

Targeted messages uses HPKE to encrypt the message content between two leaves.

3.2.3.1. Sender data encryption

In addition, TargetedMessageSenderAuthData is encrypted in a similar way to MLSSenderData as described in section 6.3.2 in [mls-protocol]. The TargetedMessageSenderAuthData.sender_leaf_index field is the leaf index of the sender. The TargetedMessageSenderAuthData.authentication_scheme field is the authentication scheme used to authenticate the sender. The TargetedMessageSenderAuthData.signature field is the signature of the TargetedMessageTBS structure. The TargetedMessageSenderAuthData.kem_output field is the KEM output of the HPKE encryption.

The key and nonce provided to the AEAD are computed as the KDF of the first KDF.Nh bytes of the hpke_ciphertext generated in the following section. If the length of the hpke_ciphertext is less than KDF.Nh, the whole hpke_ciphertext is used. In pseudocode, the key and nonce are derived as:

```
   sender_auth_data_secret
     = DeriveExtensionSecret(extension_secret, "targeted message sender auth data
")

   ciphertext_sample = hpke_ciphertext[0..KDF.Nh-1]

   sender_data_key = ExpandWithLabel(sender_auth_data_secret, "key",
                         ciphertext_sample, AEAD.Nk)
   sender_data_nonce = ExpandWithLabel(sender_auth_data_secret, "nonce",
                         ciphertext_sample, AEAD.Nn)
```

The Additional Authenticated Data (AAD) for the SenderAuthData ciphertext is the first three fields of TargetedMessage:

```
   struct {
     opaque group_id<V>;
     uint64 epoch;
     uint32 recipient_leaf_index;
   } SenderAuthDataAAD;
```

3.2.3.2. Padding

The TargetedMessage structure does not include a padding field. It is the responsibility of the sender to add padding to the message as used in the next section.

3.2.4. Authentication

For ciphersuites that support it, HPKE mode_auth_psk is used for authentication. For other ciphersuites, HPKE mode_psk is used along with a signature. The authentication scheme is indicated by the authentication_scheme field in TargetedMessageContent. See Section 3.2.5 for more information.

For the PSK part of the authentication, clients export a dedicated secret:

```
   targeted_message_psk
     = DeriveExtensionSecret(extension_secret, "targeted message psk")
```

The functions SealAuth and OpenAuth defined in [hpke] are used as described in Section 2.1.3 with an empty context. Other functions are defined in [mls-protocol].

3.2.4.1. Authentication with HPKE

The sender MUST set the authentication scheme to TargetedMessageAuthScheme.HPKEAuthPsk.

As described in Section 2.1.3 the hpke_context is a LabeledExtensionContent struct with the following content, where group_context is the serialized context of the group.

```
label = "MLS 1.0 ExtensionData"
extension_type = ExtensionType
extension_data = group_context
```

The sender then computes the following:

```
(kem_output, hpke_ciphertext) = SealAuthPSK(receiver_node_public_key,
                                            hpke_context,
                                            targeted_message_tbm,
                                            message,
                                            targeted_message_psk,
                                            psk_id,
                                            sender_node_private_key)
```

The recipient computes the following:

```
message = OpenAuthPSK(kem_output,
                      receiver_node_private_key,
                      hpke_context,
                      targeted_message_tbm,
                      hpke_ciphertext,
                      targeted_message_psk,
                      psk_id,
                      sender_node_public_key)
```

3.2.4.2. Authentication with signatures

The sender MUST set the authentication scheme to TargetedMessageAuthScheme.SignatureHPKEPsk. The signature is done using the signature_key of the sender’s LeafNode and the corresponding signature scheme used in the group.

The sender then computes the following with hpke_context defined as in Section 3.2.4.1:

```
(kem_output, hpke_ciphertext) = SealPSK(receiver_node_public_key,
                                        hpke_context,
                                        targeted_message_tbm,
                                        message,
                                        targeted_message_psk,
                                        epoch)
```

The signature is computed as follows, where the extension_type is the type of this extension (see Section 4).

```
   signature = SafeSignWithLabel(extension_type, ., "TargetedMessageTBS", targete
d_message_tbs)
```

The recipient computes the following:

```
   message = OpenPSK(kem_output,
                     receiver_node_private_key,
                     hpke_context,
                     targeted_message_tbm,
                     hpke_ciphertext,
                     targeted_message_psk,
                     epoch)
```

The recipient MUST verify the message authentication:

```
   SafeVerifyWithLabel.verify(extension_type,
                           sender_leaf_node.signature_key,
                           "TargetedMessageTBS",
                           targeted_message_tbs,
                           signature)
```

3.2.5. Guidance on authentication schemes

If the groups ciphersuite does not support HPKE mode_auth_psk, implementations MUST choose TargetedMessageAuthScheme.SignatureHPKEPsk.

If the groups ciphersuite does support HPKE mode_auth_psk, implementations CAN choose TargetedMessageAuthScheme.HPKEAuthPsk if better efficiency and/or repudiability is desired. Implementations SHOULD consult [hpke-security-considerations] beforehand.

3.3. Content Advertisement

3.3.1. Description

This section describes two extensions to MLS. The first allows MLS clients to advertise their support for specific formats inside MLS application_data. These are expressed using the extensive IANA Media Types registry (formerly called MIME Types). The accepted_media_types LeafNode extension lists the formats a client supports inside application_data. The second, the required_media_types GroupContext extension specifies which media types need to be supported by all members of a particular MLS group. These allow 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 insure that the membership and policies are compatible.

Finally, this document defines a minimal framing format so MLS clients can signal which media type is being sent when multiple formats are permitted in the same group. 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.

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

3.3.2. 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. 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 */
    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 {
    MediaType media_types<V>;
} MediaTypeList;

MediaTypeList accepted_media_types;
MediaTypeList required_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.

3.3.3. Expected Behavior

An MLS client which implements this section SHOULD include the accepted_media_types extension in its LeafNodes, listing all the media types it can receive. As usual, the client also includes accepted_media_types in its capabilities field in its LeafNodes (including LeafNodes inside its KeyPackages).

When creating a new MLS group for an application using this specification, the group MAY include a required_media_type extension in the GroupContext Extensions. As usual, the client also includes required_media_types in its capabilities field in its LeafNodes (including LeafNodes inside its KeyPackages). When used in a group, the client MUST include the required_media_types and accepted_media_types extensions in the list of extensions in RequiredCapabilities.

MLS clients SHOULD NOT add an MLS client to an MLS group with required_media_types unless the MLS client advertises it can support all of the required MediaTypes. As an exception, a client could be preconfigured to know that certain clients support the requried 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.

3.3.4. Framing of application_data

When an MLS group contains the required_media_types GroupContext extension, 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 required_media_types.

3.4. 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 extension type. It is designed to be included in External Commits.

3.4.1. Extension 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 extension in their capabilities; such a group is a "self-remove-capable group".

An MLS client which supports this extension 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.

3.5. Last resort KeyPackages

Type: KeyPackage extension

3.5.1. Description

Section 10 of [RFC9420] details that clients are required to pre-publish KeyPackages s.t. 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 extension 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 extension 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 an extension rather than communicating the information out-of-band is that the extension 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.

3.5.2. Format

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

As a result, a LastResort Extension contains the ExtensionType with an empty extension_data field.

4. 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.

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

4.1. MLS Wire Formats

4.1.1. MLS Extension Message

* Value: 0x0006

* Name: mls_extension_message

* Recommended: Y

* Reference: RFC XXXX

4.2. MLS Extension Types

4.2.1. targeted_messages_capability MLS Extension

The targeted_messages_capability MLS Extension Type is used in the capabilities field of LeafNodes to indicate the support for the Targeted Messages Extension. The extension does not carry any payload.

* Value: 0x0006

* Name: targeted_messages_capability

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

* Recommended: Y

* Reference: RFC XXXX

4.2.2. targeted_messages MLS Extension

The targeted_messages MLS Extension Type is used inside GroupContext objects. It indicates that the group supports the Targeted Messages Extension.

* Value: 0x0007

* Name: targeted_messages

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

* Recommended: Y

* Reference: RFC XXXX

4.2.3. accepted_media_types MLS Extension

The accepted_media_types MLS Extension Type is used inside LeafNode objects. It contains a MediaTypeList representing all the media types supported by the MLS client referred to by the LeafNode.

* Value: 0x0008

* Name: accepted_media_types

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

* Recommended: Y

* Reference: RFC XXXX

4.2.4. required_media_types MLS Extension

The required_media_types MLS Extension Type is used inside GroupContext objects. It contains a MediaTypeList representing the media types which are mandatory for all MLS members of the group to support.

* Value: 0x0009

* Name: required_media_types

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

* Recommended: Y

* Reference: RFC XXXX

4.2.5. last_resort_key_package MLS Extension

The last_resort_key_package MLS Extension Type is used inside KeyPackage objects. It marks the KeyPackage for usage in last resort scenarios and contains no additional data.

* Value: 0x0009

* Name: last_resort_key_package

* Message(s): KP: This extension may appear in KeyPackage objects

* Recommended: Y

* Reference: RFC XXXX

4.3. MLS Proposal Types

4.3.1. Extension Proposal

* Value: 0x0008

* Name: extension_proposal

* Recommended: Y

* Path Required: N

* External Sender: N

* Reference: RFC XXXX

4.3.2. Extension Path Proposal

* Value: 0x0009

* Name: extension_path_proposal

* Recommended: Y

* Path Required: Y

* External Sender: N

* Reference: RFC XXXX

4.3.3. Extension External Proposal

* Value: 0x000a

* Name: extension_external_proposal

* Recommended: Y

* Path Required: N

* External Sender: Y

* Reference: RFC XXXX

4.3.4. AppAck Proposal

* Value: 0x000b

* Name: app_ack

* Recommended: Y

* Path Required: Y

* Reference: RFC XXXX

4.3.5. 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: 0x000c

* Name: self_remove

* Recommended: Y

* External: N

* Path Required: Y

4.4. MLS Credential Types

4.4.1. Extension Credential

* Value: 0x0000
* Name: extension_credential
* Recommended: Y
* Reference: RFC XXXX

4.5. MLS Signature Labels

4.5.1. Labeled Extension Content

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

5. Security considerations

5.1. AppAck

TBC

5.2. Targeted Messages

In addition to the sender authentication, Targeted Messages are authenticated by using a preshared key (PSK) between the sender and the recipient. The PSK is exported from the group key schedule using the label "targeted message psk". This ensures that the PSK is only valid for a specific group and epoch, and the Forward Secrecy and Post-Compromise Security guarantees of the group key schedule apply to the targeted messages as well. The PSK also ensures that an attacker needs access to the private group state in addition to the HPKE/signature’s private keys. This improves confidentiality guarantees against passive attackers and authentication guarantees against active attackers.

5.3. Content Advertisement

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

5.4. 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.

6. References

6.1. Normative References

[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>.

6.2. Informative References

[hpke] "Hybrid Public Key Encryption", n.d., <https://www.rfc-editor.org/rfc/rfc9180.html](https://www.rfc-editor.org/rfc/rfc9180.html>.

[hpke-security-considerations] "HPKE Security Considerations", n.d., <https://www.rfc-editor.org/rfc/rfc9180.html#name-key-compromise-impersonatio](https://www.rfc-editor.org/rfc/rfc9180.html#name-key-compromise-impersonatio>.

[mls-protocol] "The Messaging Layer Security (MLS) Protocol", n.d., <https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/](https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/>.

[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>.

Contributors

Joel Alwen
Amazon
Email: alwenjo@amazon.com

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

Rohan Mahy
Wire
Email: rohan@wire.com

Marta Mularczyk
Amazon
Email: mulmarta@amazon.com

Author’s Address

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

MLS
Internet-Draft
Intended status: Informational
Expires: 24 April 2024

R. Mahy
Wire

22 October 2023

KeyPackage Context Extension for Message Layer Security (MLS)
draft-mahy-mls-kp-context-00

Abstract

This document describes a Message Layer Security (MLS) KeyPackage extension to convey a specific context or anticipated use for the KeyPackage. It is useful when a client provides the KeyPackage out-of-band to another client, and wants the specific KeyPackage used only in the anticipated context, for example a specific MLS group.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mahy-mls-kp-context/.

Discussion of this document takes place on the MLS 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/rohan-wire/mls-kp-context/.

Status of This Memo

Copyright Notice

Table of Contents

1. Terminology

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.

The terms MLS client, MLS group, LeafNode, GroupContext, KeyPackage, GroupContextExtensions Proposal, Credential, CredentialType, and RequiredCapabilities have the same meanings as in the MLS protocol [I-D.ietf-mls-protocol].

2. Introduction

In some use cases of MLS, a client might wish to provide a KeyPackage to another client, but communicate that the specific KeyPackage is only to be used in a specific context, for example to join a specific MLS group. This document describes a KeyPackage extension that can convey that context.

3. Extension Description

This document specifies a KeyPackage MLS extension kp_context of type ContextPair. The syntax is described using the TLS Presentation Language [RFC8446]

Each PerDomainTrustAnchor represents a specific identity domain which is expected and authorized to participate in the MLS group. It contains the domain name and the specific trust anchor used to validate identities for members in that domain.

```
enum {
  reserved(0),
  groupid(1),
  uri(2),
  domain(3),
  jwk_thumbprint(4)
  (255)
} ContextType;

struct {
    ContextType context_type;
    opaque context_value<V>;
} ContextPair;

ContextPair kp_context;
```

4. IANA Considerations

This document proposes registration of a new MLS Extension Type.

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

4.1. kp_context MLS Extension Type

The kp_context MLS Extension Type is used inside KeyPackage objects. It contains a URN Anchors object representing the trust anchors which are expected for identity validation inside the MLS group.

```
  Template:
  Value: 0x000B
  Name: kp_context
  Message(s): This extension may appear in KeyPackage objects
  Recommended: Y
  Reference: RFC XXXX
```

4.2. urn:ietf:mls:kp_context:group_id URN registration

Namespace Identifier: Requested of IANA (formal) or assigned by IANA (informal).

Version: 1

Date: 2023-08-01

Registrant:
Rohan Mahy
rohan.ietf@gmail.com

Purpose: Described in Section 3 of RFCXXXX.

Syntax: Described in Section 3 of RFCXXXX.

Assignment: Described in Section 4.1 of RFCXXXX.

Security and Privacy: Described in Section 5 of RFCXXXX.

Interoperability: Described in Section 3 of this document.

Resolution: Described in Section 3 of this document.

Documentation: RFCXXXX

Additional Information: none

Revision Information: n/a

5. Security Considerations

The Security Considerations of MLS apply.

The use of this extension may reveal the client's intentions or wishes in an out-of-band protocol, which may have weaker privacy protections than MLS handshake messages.

6. Normative References

[I-D.ietf-mls-protocol]
Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", Work in Progress, Internet-Draft, draft-ietf-mls-protocol-20, 27 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-20>.

[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>.

[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>.

Author's Address

Rohan Mahy
Wire
Email: rohan.mahy@wire.com

MLS
Internet-Draft
Intended status: Informational
Expires: 11 January 2024

R. Mahy
M. Amiot
Wire
10 July 2023

Messaging Layer Security Ciphersuite using X25519Kyber768Draft00 Key Exchange Mechanism
draft-mahy-mls-x25519kyber768draft00-00

Abstract

This document registers a new Messaging Layer Security (MLS) ciphersuite using the hybrid post-quantum resistant / traditional (PQ/T) Key Exchange Mechanism X25519Kyber768Draft00.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mahy-mls-x25519kyber768draft00/.

Discussion of this document takes place on the MLS 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/rohan-wire/mls-x25519kyber768draft00/.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document reserves a Messaging Layer Security (MLS) [I-D.ietf-mls-protocol] ciphersuite value based on the MLS default ciphersuite, but replacing the KEM with the hybrid post-quantum / traditional Key Exchange Mechanism X25519Kyber768Draft00 [I-D.draft-westerbaan-cfrg-hpke-xyber768d00] which was assigned the Hybrid Public Key Encryption (HPKE) Key Exchange Mechanism (KEM) Identifier value 0x0030.

2. Security Considerations

This ciphersuite uses a hybrid post-quantum/traditional KEM and a traditional signature algorithm. As such, it is designed to provide confidentiality against quantum and classical attacks, but provides authenticity against classical attacks only. This is actually very useful, because an attacker could store MLS-encrypted traffic that uses any classical KEM today. If years or decades in the future a quantum attack on classical KEMs becomes feasible, the traffic sent today (some of which could still be sensitive in the future) will then be readable. By contrast, an attack on a signature algorithm in MLS would require an active attack which can extract the private key during the signature key's lifetime.

The security properties of [I-D.draft-westerbaan-cfrg-hpke-xyber768d00] apply.

3. IANA Considerations

This document registers a new MLS Ciphersuite value.

```
Value:     0x0030 (please)
Name:      MLS_128_X25519Kyber768Draft00_AES128GCM_SHA256_Ed25519
Required:  N
Reference: This document
```

4. Normative References

[I-D.draft-westerbaan-cfrg-hpke-xyber768d00]
Westerbaan, B. and C. A. Wood, "X25519Kyber768Draft00 hybrid post-quantum KEM for HPKE", Work in Progress, Internet-Draft, draft-westerbaan-cfrg-hpke-xyber768d00-02, 4 May 2023, <https://datatracker.ietf.org/doc/html/draft-westerbaan-cfrg-hpke-xyber768d00-02>.

[I-D.ietf-mls-protocol]
Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", Work in Progress, Internet-Draft, draft-ietf-mls-protocol-20, 27 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-20>.

Acknowledgments

Thanks to Joël Alwen, Marta Mularczyk, and Britta Hale.

Authors' Addresses

Rohan Mahy
Wire
Email: rohan.mahy@wire.com

Mathieu Amiot
Wire
Email: mathieu.amiot@wire.com