Workgroup: Network Working Group
Internet-Draft: draft-ietf-mls-extensions-01
Published: 13 March 2023
Intended Status: Informational
Expires: 14 September 2023
Authors: R. Robert
Phoenix R&D

The Messaging Layer Security (MLS) Extensions

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

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

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

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

This Internet-Draft will expire on 14 September 2023.

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-01

- *Add Content Advertisement extensions

draft-00

- *Initial adoption of draft-robert-mls-protocol-00 as a WG item.
- *Add Targeted Messages extension (*)

2. Extensions

2.1. AppAck

Type: Proposal

2.1.1. 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.

2.2. Targeted messages

2.2.1. 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 reuse mechanisms from [mls-protocol], in particular [hpke].

2.2.2. Format

This extensions introduces a new message type to the MLS protocol, TargetedMessage in WireFormat and MLSMessage:

```
enum {
  ...
  mls_targeted_message(6),
  ...
  (255)
} WireFormat;

struct {
    ProtocolVersion version = mls10;
    WireFormat wire_format;
    select (MLSMessage.wire_format) {
        ...
        case mls_targeted_message:
            TargetedMessage targeted_message;
    }
} MLSMessage;
```

The TargetedMessage message type is defined as follows:

```
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.

2.2.3. Encryption

Targeted messages use HPKE to encrypt the message content between two leaves. The HPKE keys of the LeafNode are used to that effect, namely the encryption_key field.

In addition, TargetedMessageSenderAuthData is encrypted in a similar way to MLSSenderData as described in section 7.3.2 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
  = MLS-Exporter("targeted message sender auth data", "", KDF.Nh)

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;
```

2.2.3.1. 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.

2.2.4. 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 2.2.5 for more information.

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

```
targeted_message_psk = MLS-Exporter("targeted message psk", "", KDF.Nh)
```

Th functions SealAuth and OpenAuth are defined in [hpke]. Other functions are defined in [mls-protocol].

2.2.4.1. Authentication with HPKE

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

The sender then computes the following:

```
(kem_output, hpke_ciphertext) = SealAuthPSK(receiver_node_public_key,
                                            group_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,
                      group_context,
                      targeted_message_tbm,
                      hpke_ciphertext,
                      targeted_message_psk,
                      psk_id,
                      sender_node_public_key)
```

2.2.4.2. 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:

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

signature = SignWithLabel(., "TargetedMessageTBS", targeted_message_tbs)
```

The recipient computes the following:

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

The recipient MUST verify the message authentication:

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

2.2.5. Guidance on authentication schemes

If the group's ciphersuite does not support HPKE mode_auth_psk, implementations MUST choose TargetedMessageAuthScheme.SignatureHPKEPsk.

If the group's 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.

2.3. Content Advertisement

2.3.1. 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]).

2.3.2. 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.

2.3.3. 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.

2.3.4. 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. 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

3.1. MLS Wire Formats

3.1.1. Targeted Messages wire format

*Value: 0x0006

*Name: * Name: mls_targeted_message

*Recommended: Y

*Reference: RFC XXXX

3.2. MLS Extension Types

3.2.1. 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

3.2.2. 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

3.2.3. 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

3.2.4. 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

3.3. MLS Proposal Types

3.3.1. AppAck Proposal

- *Value: 0x0008
- *Name: app_ack
- *Recommended: Y
- *Path Required: Y
- *Reference: RFC XXXX

4. Security considerations

4.1. AppAck

TBC

4.2. 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.

4.3. 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. Contributors

The accepted_media_types and rejected_media_types extensions were written by Rohan Mahy.

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>.

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>.

Author's Address

Raphael Robert
Phoenix R&D

Email: ietf@raphaelrobert.com