Network Working Group
Internet-Draft
Intended status: Informational
Expires: 9 October 2025

K. Kohbrok
R. Robert
Phoenix R&D

7 April 2025

MIMI Metadata Minimalization (MIMIMI)
draft-kohbrok-mimi-metadata-minimalization-02

Abstract

This document describes a proposal to run the MIMI protocol in a way that reduces the ability of the Hub and service providers to associate messaging activity of clients with their respective identities.

For now, this document only contains a high-level description of the mechanisms involved.

Discussion Venues

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

Discussion of this document takes place on the More Instant Messaging Interoperability Working Group mailing list (mimi@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/mimi/.

Source for this draft and an issue tracker can be found at https://github.com/kkohbrok/mimi-metadata-minimalization.

Status of This Memo

Table of Contents

1. Introduction

Since the MIMI protocol uses MLS PublicMessages for Proposals and Commits, both Hub and Provider can observe and track activities of clients both within an individual group and across groups.

The goal of the scheme described in this document are as follows:

* For a given client, prevent the Hub and all service providers from associating the activity of the client with that client's identifier.

* For a given client, prevent the Hub and all service providers except for the client's service provider from associating the activity of that client in one group from that in any other.

These propoerties are achieved withouth limiting the ability of the Hub to track a group's group state and provider public group information to newly joining clients.

However, the scheme comes with a few operative requirements and the following general limitations:

* There needs to be the notion of a "connection" between two users. Users that are connected can link a pseudonym with a user's real identity and can (selectively) allow others to do the same.

* Only connected users can add one-another to groups.

* Users that join a group need help of an existing group member in linking the pseudonyms visible in the group to real user identities.

This document only contains a high-level description of the individual changes required to achieve the goals detailed above.

2. Pseudonymization

The main mechanism to hide the user and client identifiers from the hub and providers in the context of individual groups and use user-level and client-level pseudonyms instead.

The purpose of the pseudonymization scheme described in this document is to hide metadata from service providers with as little impact as possible on the features provided by the base MIMI protocol. This means that by default, User and client identifiers are still visible to (other) users and their clients.

In addition, the pseudonymization scheme can also be used to hide a user's identity from other users. See Section Section 6 for more details.

3. Pseudonymity through credential encryption

Clients use PseudonymousCredentials to represent them in MIMI rooms to hide their identities from providers

```
struct {
  IdentifierUri client_pseudonym;
  IdentifierUri user_pseudonym;
  opaque signature_public_key;
  opaque identity_link_ciphertext<V>;
} PseudonymousCredential
```

PseudonymousCredentials don’t contain a client’s client and user ID. Instead, they use two pseudonyms: a user pseudonym shared by all other PseudonymousCredentials of the user’s clients in a given room and a client pseudonym unique for each PseudonymousCredential. Each pseudonym is an IdentifierUri made up of a randomly generated UUID and the user’s provider’s domain.

```
struct {
  IdentifierUri client_pseudonym;
  IdentifierUri user_pseudonym;
  opaque signature_public_key;
} PseudonymousCredentialTBS

struct {
  /* SignWithLabel(., "PseudonymousCredentialTBS",
    PseudonymousCredentialTBS) */
  opaque pseudonymous_credential_signature<V>;
  Credential client_credential;
} IdentityLinkTBE
```

The identity_link_ciphertext is created by encrypting the IdentityLinkTBE, which contains the client’s real credential, as well as a signature over the PseudonymousCredentialTBS using the client credential’s signature key.

The identity_link_key used for encryption is unique per pseudonymous credential. It is derived from the client’s connection_key.

```
identity_link_key = ExpandWithLabel(connection_key,
  "IdentityLinkKey", PseudonymousCredentialTBS, KDF.Nh)
```

See Section 5 for more details on the connection_key.

Pseudonyms are specific to each room and client since the identity_link_key is unique per pseudonymous credential and pseudonymous credentials are used in only once (i.e. in one room).

4. Identity link key management in rooms

All clients in a room MUST be able to decrypt the identity_link_ciphertext of all other clients in the room, except in the cases detailed in Section 6.

TODO: The following scheme should be replaced by TreeWrap for FS and PCS. It’s a simple placeholder for now.

The hub holds encrypted copies of the identity_link_keys of all clients in the room and updates them as clients get added, removed and updated. The identity_link_keys are encrypted with the room's identity_link_wrapper_key.

The identity_link_wrapper_key is freshly generated by the room's creator and does not change throughout the lifetime of the group.

The identity_link_wrapper_key is distributed to new participants as they join the room.

4.1. Add flow

If a group member adds a new user, the adder MUST include a IdentityLinkExtension in the Welcome's GroupInfoExtensions.

```
struct {
  opaque identity_link_wrapper_key<V>;
} IdentityLinkExtension
```

The adder MUST also include the encrypted identity_link_key of all added users in the AAD of the commit message. The order of the encrypted identity_link_keys in the AAD MUST match the order of the Add proposals in the Commit message.

Note that this requires the adder to have a connection with each added user as specified in Section 5.

TODO: Note here that the sender MUST use the SafeAAD as defined in the extension doc.

```
struct {
  opaque encrypted_identity_link_keys<V>;
} AddAAD
```

Existing group members can then decrypt the identity_link_keys to learn the real identities of the added users.

When it receoves a Commit message, the hub adds the encrypted identity_link_keys to the room's state.

Finally, the added user receiving the welcome can then obtain the encrypted identity_link_keys from the hub and decrypt them using the identity_link_wrapper_key from the extension and use the decrypted identity_link_keys to decrypt the identity_link_ciphertexts of all other clients in the room.

4.2. Join flow

Members joining a room through the join flow need to obtain the identity_link_wrapper_key out of band. The key can be shared any other group member.

TODO: When defining features such as join/invitation links the identity_link_wrapper_key should be included in the link.

When joining the group, the joining client MUST include its own encrypted identity_link_key in the AAD of the external commit message via the AddAAD struct.

5. Connections

A connection is a mutually agreed-upon relation betwen two users. Users can establish connections between one-another through the following process.

Let’s call the initiator of the connection establishment Alice and the responding user Bob.

At the time of registration with their local provider, Alice and Bob generate an HPKE keypair and upload the public key (called the connection establishment public key) to their respective providers. The providers make the keys available to all other users without requiring authentication.

To hide Alice’s identity from Bob’s provider throughout the connection establishment process, Alice obtains Bob’s connection establishment public key from Bob’s provider.

Alice now creates a new conversation with her local provider as hub and adds all of her clients to that conversation. She then compiles the following information into a connection request:

```
struct {
  Credential requester_client_credential;
  opaque connection_room_id<V>;
  opaque connection_room_identity_link_wrapper_key<V>;
  opaque requester_connection_key<V>;
  opaque connection_response_key<V>;
} ConnectionRequest
```

The connection_response_key is a fresh HPKE keypair that Alice generates when she creates the connection request. It is later used by Bob to encrypt his own connection_key if Bob chooses to accept the request.

TODO: The request will likely contain more information such as e.g. an authorization token that allows Bob to fetch Alice's KeyPackages. Any such information must also be included in the ConnectionResponse below.

Alice then encrypts the connection request using Bob's connection establishment public key and sends it to Bob's provider.

Bob fetches the connection request from his provider and decrypts it using his connection establishment key.

He can inspect Alice's requester_client_credential and decide whether to accept the request.

If Bob accepts, he creates a connection response:

```
struct {
  opaque responder_connection_key<V>;
} ConnectionResponse
```

Bob then follows the join flow for the room with the connection_room_id, including the ConnectionResponse encrypted under the connection_response_key in the join message's (i.e. the external commit's) AAD. As part of that process, he obtains the encrypted identity_link_keys for Alice's clients and verifies that they belong to the same user as the requester_client_credential in the connection request.

Bob joining the room signals to Alice that Bob has accepted her connection request. Alice can decrypt Bob's ConnectionResponse in the join message's AAD using the connection response key she initially included in the connection request.

Both Alice and Bob now share a room and can derive identity_link_keys for each other's clients, which in turn allows them to add each other to rooms.

6. User-to-user pseudonymity

In some cases, users may want to communicate with each other without revealing their real identities to one-another. Room policy can thus specify the extent to which clients are required to provide identity_link_keys. For example, identity_link_keys might not be allowed at all, or only admins are required to provide identity_link_keys.

7. KeyPackages

The use of user and client pseudonyms requires some additional care when generating and uploading KeyPackages.

The user and client pseudonyms in the PseudonymousCredential must be unique across groups. For the client pseudonym to be unique across groups, clients can generate it randomly for each KeyPackage.

However, to allow the hub to associate a set of leaves with a user, the user pseudonym must be consistent across the leaves in that group. This means that clients need to coordinate to provide KeyPackage sets, where each KeyPackage in a set contains a consistent user pseudonym and no two sets contain the same pseudonym. Each such set can be used in the context of one group.

A procedure for a client to upload KeyPackages could thus look as follows:

1. Check for which user pseudonyms other clients of the same user have uploaded KeyPackages.

2. Identify user pseudonyms for which the KeyPackage set is missing a KeyPackage of this client.

3. Generate and upload KeyPackages for those user pseudonyms.

4. Upload as many KeyPackages as desired, thus starting new KeyPackage sets for other clients to complete.

At least one KeyPackage set of each user must consist of last-resort KeyPackages. If such a last-resort set is used more than once, the use of the same user pseudonym allows the hub to learn that one user is active in two particular groups. The re-use of such a set cannot be avoided entirely, but users upload more than one set (and re-upload last-resort sets frequently) to mitigate this at least to a certain degree.

When uploading KeyPackages, user generally upload KeyPackages to their provider indexed by their connection token. To avoid leaking metadata to the user’s own provider, the upload must not be connected to the user’s identity.

Users can then fetch KeyPackages of connected users from their respective providers using the connected user’s connection token.

8. Message Fanout

This protocol assumes that clients have one or more queue IDs that their local provider can use to route incoming messages to a place that the client can access.

For each member of a given group, the member’s queue ID is stored in a leaf node extension s.t. the hub can forward that information to the client’s local provider upon fanout.

Since in most cases, there will only be a single queue for each client (as opposed to one queue per group), the queue ID is encrypted using an HPKE key of the client’s provider.

When receiving the encrypted queue ID with the message fanout, the provider can decrypt the queue ID and route the message to the correct queue.

Authors’ Addresses

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

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