White Paper
SKISSM

2021.09.08 version 1.1

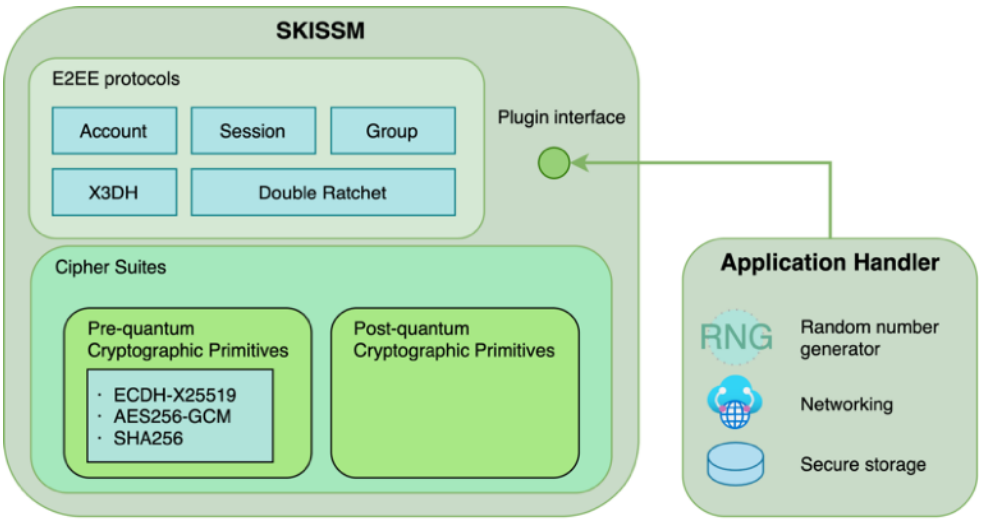
Academia Sinica
中央研究院

資訊科技創新研究中心
Research Center for Information Technology Innovation

Contents

Introduction

This white paper provides a technical overview on the end-to-end encryption (E2EE) protocols and security aspects implemented by SKI software security module (SKISSM).



SKISSM software infrastructure

SKISSM implements the Signal protocol [1][2][3][4] and aims to make it easy to build versatile end-to-end messaging applications. The two crucial security properties provided:

- End-to-end encryption

  Only sender and recipient (and not even the server) can decrypt the content.

- Forward secrecy

  Past sessions are protected against future compromises of keys or passwords.

Abbreviations

- **ck:** chain key
- **C**: ciphertext
- **Dec(x, y)**: decrypt message x with key y using AES256 with GCM mode
- **ECDH(x, y)**: elliptic curve Diffie-Hellman key exchange with X25519 algorithm where x is a private key and y is a public key
- **Enc(x, y)**: encrypt message x with key y using AES256 with GCM mode
- **ek, ek^{-1}**: ephemeral key pair
- **HKDF(IKM, salt, info)**: HKDF with SHA-256 with input key material IKM, salt, and info
- **HMAC(key, input)**: HMAC with SHA-256 with the key and the input
- **ik, ik^{-1}:** identity key pair
- **mk**: message key
- **opk, opk^{-1}**: one-time pre-key pair
- **P**: plaintext
- **rk, rk^{-1}**: ratchet key pair
- **RK**: root key
- **sig = Sign(x, y)**: sign message x with private key y and output the signature sig
- **spk, spk^{-1}**: signed pre-key pair
- **sk**: shared secret key
- **sk_priv**: signature private key
- **sk_pub**: signature public key
- **Verify(sig, k)**: verify the signature sig with the public key k

Cryptographic Algorithms

Cipher suite

The default cipher suite in SKISSM uses the following cryptographic primitives:

- ECDH-X25519
- AES256-GCM
- SHA256

A cipher suite in SKISSM is constructed by the following cryptographic function interface. SKISSM provides an implementation that utilizes the curve25519-donna[10] and mbed TLS[11] library.

- gen_private_key /*Generate a random private key*/
- gen_public_key /*Generate a public key by given private key*/
- gen_key_pair /*Generate a random key pair*/
- dh /* Calculate shared secret by Diffie–Hellman (DH) algorithm */
- encrypt /*Encrypt a given plain text*/
- decrypt /*Decrypt a given cipher text*/
- sign /*Sign a message*/
- verify /*Verify a signature with given message*/
- hkdf /*HMAC-based key derivation function*/
- hmac /*Keyed-Hashing for message authentication*/
- hash /*Hash function*/

Algorithms

Here is the list of algorithms implemented by SKISSM.

Outbound session

An outbound session is used to encrypt one-to-one message for sending to remote peer. To build a new outbound session, Alice first acquires Bob's pre-key bundle from messaging server, then performs the following steps:

- Verify(Sig, ik_B)
- Generate ek_A (32 bytes key pair) and rk_A (32 bytes key pair)
- Calculate share secret using X3DH

  k_1(32 bytes) = ECDH(ik_A^{-1}, spk_B)

  k_2(32 bytes) = ECDH(ek_A^{-1}, ik_B)

  k_3(32 bytes) = ECDH(ek_A^{-1}, spk_B)

  k_4(32 bytes) = ECDH(ek_A^{-1}, opk_B)

  secret(128 bytes) = k_1 ll k_2 ll k_3 ll k_4

  sk(64 bytes) = HKDF(secret, salt[32]={0}, info="ROOT")

To encrypt message by using established outbound session:

- Apply the Double Ratchet Algorithm

  RK(32 bytes) = prefix 32 bytes of sk

  The first ratchet key is just the Bob's signed pre-key. That is, rk_B = spk_B

  secret_input(32 bytes) = ECDH(rk_A^{-1}, rk_B)

  Next sk(64 bytes)

  = HKDF(secret_input, salt=RK, info="RATCHET")

  = next RK(32 bytes) ll sender_chain_key(32 bytes)

  mk(48 bytes)

  = HKDF(sender_chain_key, salt[32]={0}, info="MessageKeys")

  ciphertext = Enc(plaintext, mk)

- Send the ciphertext and rk_A to Bob

Inbound session

An inbound session is used to decrypt one-to-one message received from remote peer. When Bob received a pre-key message from Alice, Bob can build a new inbound session by performing the following steps:

- Calculate share secret using X3DH

  $k_1=ECDH(spk_B^{-1}, ik_A)$

  $k_2=ECDH(ik_B^{-1}, ek_A)$

  $k_3=ECDH(spk_B^{-1}, ek_A)$

  $k_4=ECDH(opk_B^{-1}, ek_A)$

  secret(128 bytes) = k_1 ll k_2 ll k_3 ll k_4

  sk(64 bytes) = HKDF(secret, salt[32]={0}, info="ROOT")

To decrypt message by using established inbound session:

- Apply the Double Ratchet Algorithm

  RK(32 bytes) = prefix 32 bytes of sk

  secret_input(32 bytes) = $ECDH(rk_B^{-1}, rk_A)$

  next sk(64 bytes)

  = HKDF(secret_input, salt=RK, info="RATCHET")

  = next RK(32 bytes) ll receiver_chain_key(32 bytes)

  mk(48 bytes)

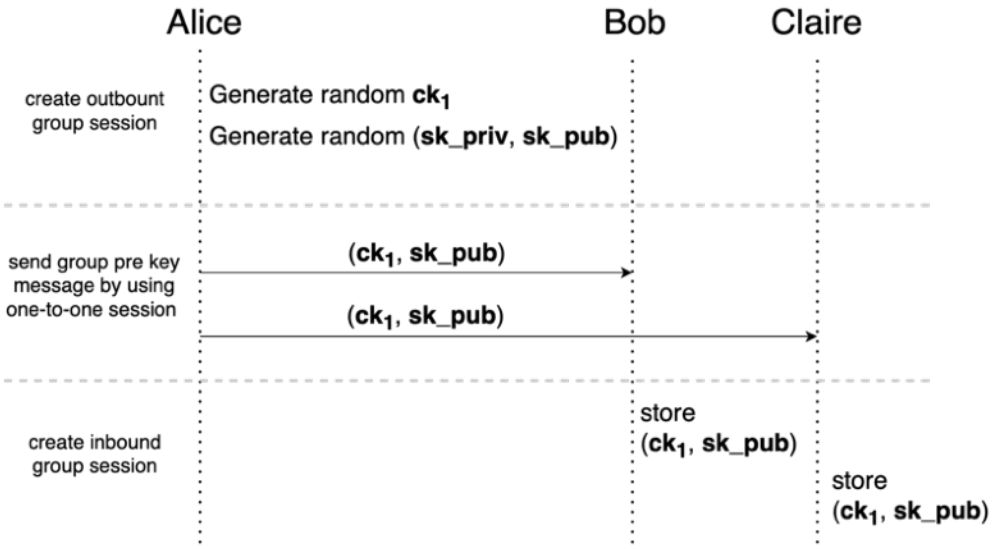
  = HKDF(receiver_chain_key, salt[32]={0}, info="MessageKeys")

  plaintext = Dec(cipertext, mk)

Group session

Each group member creates an outbound group session for encrypting and sending group message. On the other hand, the other group members create inbound group session with respect to the outbound group session for decrypting received group message.

- Group creator creates an outbound group session by generating random ck, and random (sk_priv, sk_pub)
- Group creator then send group pre-key (ck, sk_pub) to each group member by using one-to-one session. Each group member can build inbound group session by using group pre-key.



Group members alteration

When some group members are added or removed, the group member who makes the changed event first rebuild the outbound group session, then notify the affected group members to rebuild the corresponding inbound group session, and subsequently rebuild each group member's outbound group session. As a result, all outbound and inbound group sessions will be renewed, and the removed group members has no information about the updated group sessions.

Group message

To encrypt and send outbound group message, Alice uses the established outbound group session and performs the following steps:

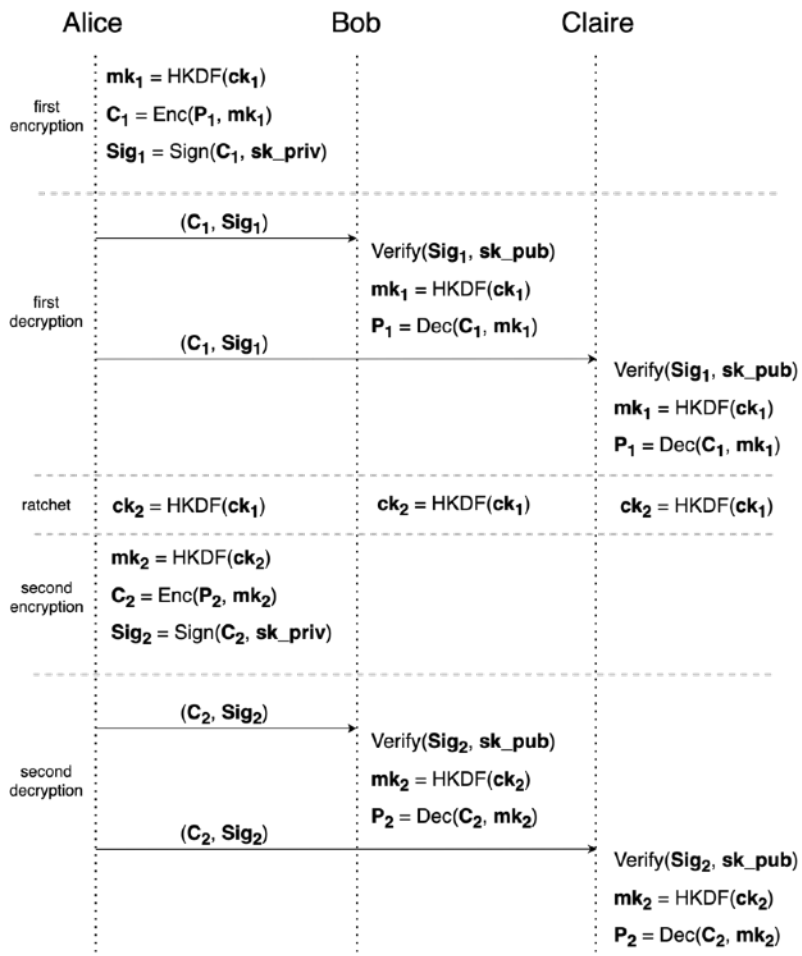
- mk = HKDF(ck)
- cipertext = Enc(plaintext, mk)
- Sig = Sign(cipertext, sk_priv)
- Send (cipertext, Sig) to each group member

Alice uses the outbound group session to ratchet ck for the next encryption.

To decrypt a received inbound group message, Bob and other group members use the established inbound group session and perform the following steps:

- Verify(Sig, sk_pub)
- mk = HKDF(ck)
- plaintext = Dec(cipertext, mk)

Each group member uses their own inbound group session to ratchet ck for the next decryption.



Protocol Data Types

Here is the complete list of protocol data types that will be used by SKISSM. These protocol data types are created by using use Protocol Buffers version 3 (proto3[5]).

Common Data Types

```
e2ee_address {
        bytes domain = 1;
        bytes user_id = 2;
        bytes device_id = 3;
        bytes group_id = 4;
}
```

```
message key_pair {
   bytes public_key = 1;
   bytes private_key = 2;
}
```

```
message signed_pre_key_pair {
   uint32 spk_id = 1;
   key_pair key_pair = 2;
   bytes signature = 3;
   uint64 ttl = 4;
}
```

```
message signed_pre_key_public {
   uint32 spk_id = 1;
   bytes public_key = 2;
   bytes signature = 3;
}
```

```
message one_time_pre_key_pair {
   uint32 opk_id = 1;
   bool used = 2;
   key_pair key_pair = 3;
}
```

```
message one_time_pre_key_public {
    uint32 opk_id = 1;
    bytes public_key = 2;
}
```

```
message e2ee_pre_key_bundle {
    e2ee_address peer_address = 1;
    bytes identity_key_public = 2;
    signed_pre_key_public signed_pre_key_public = 3;
    one_time_pre_key_public one_time_pre_key_public = 4;
}
```

```
message e2ee_account {
    uint32 version = 1;
    bytes account_id = 2;
    bool saved = 3;
    e2ee_address address = 4;
    key_pair identity_key_pair = 5;
    signed_pre_key_pair signed_pre_key_pair = 6;
    repeated one_time_pre_key_pair one_time_pre_keys = 7;
    uint32 next_signed_pre_key_id = 8;
    uint32 next_one_time_pre_key_id = 9;
}
```

E2EE Message Related Data Types

```
enum e2ee_message_type {
    PRE_KEY = 0;
    MESSAGE = 1;
    GROUP_MESSAGE = 2;
}
```

```
message e2ee_message {
    uint32 version = 1;
    e2ee_address from = 2;
    e2ee_address to = 3;
    bytes session_id = 4;
    e2ee_message_type msg_type = 5;
    bytes payload = 6;
}
```

```
message e2ee_pre_key_payload {
    bytes alice_identity_key = 1;
    bytes alice_ephemeral_key = 2;
    bytes bob_signed_pre_key = 3;
    bytes bob_one_time_pre_key = 4;
    e2ee_msg_payload msg_payload = 5;
}
```

```
message e2ee_msg_payload {
    uint32 sequence = 1;
    bytes ciphertext = 2;
    bytes ratchet_key = 3;
}
```

```
message e2ee_group_msg_payload {
    uint32 sequence = 1;
    bytes ciphertext = 2;
    bytes signature = 3;
}
```

```
enum e2ee_plaintext_type {
    COMMON_MSG = 0;
    GROUP_PRE_KEY = 1;
}
```

```
message e2ee_plaintext {
    uint32 version = 1;
    e2ee_plaintext_type plaintext_type = 2;
    bytes payload = 3;
}
```

```
message e2ee_group_pre_key_payload {
    uint32 version = 1;
    bytes session_id = 2;
    bytes old_session_id = 3;
    e2ee_address group_address = 4;
    repeated e2ee_address member_addresses = 5;
    uint32 sequence = 6;
    bytes chain_key = 7;
    bytes signature_public_key = 8;
}
```

```
message chain_key {
    uint32 index = 1;
    bytes shared_key = 2;
}
```

```
message message_key {
    uint32 index = 1;
    bytes derived_key = 2;
}
```

E2EE Session Related Data Types

```
Message message sender_chain_node {
    key_pair ratchet_key_pair = 1;
    chain_key chain_key = 2;
}
```

```
Message message receiver_chain_node {
    bytes ratchet_key_public = 1;
    chain_key chain_key = 2;
}
```

```
Message message skipped_message_key_node {
    bytes ratchet_key_public = 1;
    message_key message_key = 2;
}
```

```
Message message e2ee_ratchet {
    bytes root_key = 1;
    sender_chain_node sender_chain = 2;
    repeated receiver_chain_node receiver_chains = 3;
    repeated skipped_message_key_node skipped_message_keys = 4;
}
```

```
message message e2ee_session {
    uint32 version = 1;
    bytes session_id = 2;
    e2ee_address session_owner = 3;
    e2ee_address from = 4;
    e2ee_address to = 5;
    e2ee_ratchet ratchet = 6;
    bytes alice_identity_key = 7;
    bytes alice_ephemeral_key = 8;
    bytes bob_signed_pre_key = 9;
    uint32 bob_signed_pre_key_id = 10;
    bytes bob_one_time_pre_key = 11;
    uint32 bob_one_time_pre_key_id = 12;
    bool responded = 13;
    bytes associated_data = 14;
}
```

```
message message e2ee_group_session {
    uint32 version = 1;
    bytes session_id = 2;
    e2ee_address session_owner = 3;
    e2ee_address group_address = 4;
    repeated e2ee_address member_addresses = 5;
    uint32 sequence = 6;
    bytes chain_key = 7;
    bytes signature_private_key = 8;
    bytes signature_public_key = 9;
    bytes associated_data = 10;
}
```

E2EE Protocols Related Data Types

```
enum e2ee_commands {
    register_user = 0;
    register_user_response = 4097;
    delete_user = 2;
    delete_user_response = 4099;
    get_pre_key_bundle = 4;
    get_pre_key_bundle_response = 4101;
    publish_spk = 6;
    publish_spk_response = 4103;
    supply_opks = 8;
    supply_opks_response = 4105;
    create_group = 10;
    create_group_response = 4107;
    get_group = 12;
    get_group_response = 4109;
    add_group_members = 14;
    add_group_members_response = 4111;
    remove_group_members = 16;
    remove_group_members_response = 4113;
    e2ee_msg = 18;
    e2ee_msg_response = 4115;
    e2ee_group_msg = 20;
    e2ee_group_msg_response = 4117;
}
```

```
message e2ee_protocol_msg {
    uint32 id = 1;
    e2ee_commands cmd = 2;
    bytes payload = 3;
}
```

```
message register_user_request_payload {
    bytes user_name = 1;
    bytes identity_key_public = 2;
    signed_pre_key_public signed_pre_key_public = 3;
    repeated one_time_pre_key_public one_time_pre_keys = 4;
}
```

```
message register_user_response_payload {
    uint32 code = 1;
    string msg = 2;
    e2ee_address address = 3;
}
```

```
message delete_user_request_payload {
    e2ee_address address = 1;
}
```

```
message delete_user_response_payload {
    uint32 code = 1;
    string msg = 2;
}
```

```
message get_pre_key_bundle_request_payload {
    e2ee_address peer_address = 1;
}
```

```
message get_pre_key_bundle_response_payload {
    uint32 code = 1;
    string msg = 2;
    bytes user_name = 3;
    repeated e2ee_pre_key_bundle pre_key_bundle = 4;
}
```

```
message publish_spk_request_payload {
    e2ee_address user_address = 1;
    signed_pre_key_public signed_pre_key_public = 2;
}
```

```
message publish_spk_response_payload {
    uint32 code = 1;
    string msg = 2;
}
```

```
message supply_opks_request_payload {
    uint32 opks_num = 1;
    e2ee_address user_address = 2;
}
```

```
message supply_opks_response_payload {
    uint32 code = 1;
    string msg = 2;
    e2ee_address user_address = 3;
    repeated one_time_pre_key_public one_time_pre_key_public = 4;
}
```

```
message create_group_request_payload {
    e2ee_address sender_address = 1;
    repeated e2ee_address member_addresses = 2;
    bytes group_name = 3;
    e2ee_address group_address = 4;
}
```

```
message create_group_response_payload {
    uint32 code = 1;
    string msg = 2;
    e2ee_address group_address = 3;
}
```

```
message get_group_request_payload {
    e2ee_address group_address = 1;
}
```

```
message get_group_response_payload {
    uint32 code = 1;
    string msg = 2;
    bytes group_name = 3;
    repeated e2ee_address member_addresses = 4;
}
```

```
message add_group_members_request_payload {
    e2ee_address sender_address = 1;
    e2ee_address group_address = 2;
    repeated e2ee_address member_addresses = 3;
}
```

```
message add_group_members_response_payload {
    uint32 code = 1;
    string msg = 2;
}
```

```
message remove_group_members_request_payload {
    e2ee_address sender_address = 1;
    e2ee_address group_address = 2;
    repeated e2ee_address member_addresses = 3;
}
```

```
message remove_group_members_response_payload {
    uint32 code = 1;
    string msg = 2;
}
```

```
message e2ee_msg_response_payload {
    uint32 code = 1;
    string msg = 2;
}
```

```
message e2ee_group_msg_response_payload {
    uint32 code = 1;
    string msg = 2;
}
```

E2EE Protocols

SKISSM supports asynchronous and out-of-order end-to-end message encryption scheme. It also supports one-to-one messaging and group messaging for registered user with multiple devices. The complete list of protocols are elaborated below:

Register Request

In the beginning, user should send a register message with register_request_payload to server for completing the registration process. A registered peer address will be returned in the register_response_payload.

E2EE Command	Request Payload Type	Direction	Description
register_user	register_user_request_payload	outbound	Register user by providing user name, identity key, signed pre-key, and 100 onetime pre-keys.

Get Pre-key Bundle Request

In order to send encrypted message to remote peer, an outbound session will be created by first obtaining the pre-key bundle that has been published previously by remote peer. User name and an array of e2ee_pre_key_bundle (for each device published by remote peer) will be returned in the get_pre_key_bundle_response_payload.

E2EE Command	Request Payload Type	Direction	Description
get_pre_key_bundle	get_pre_key_bundle_request_payload	outbound	Get pre-key bundle by providing peer address.

Publish SPK Request

For security enhancement, spk should be renewed regularly (7 days for example) by peer user with respect to some device. The renewed spk will be published to server with this request that is automatically maintained by SKISSM conforming to the referenced period.

E2EE Command	Request Payload Type	Direction	Description
publish_spk	publish_spk_request_payload	outbound	Publish a new spk to server.

Supply OPKs Request

Messaging server is willing to keep a set of unused opks. When the rest of opks is running out, a supply opks request will be sent to respective address. SKISSM will supply 100 new opks automatically for serving this request by returning supply_opks_response_payload.

E2EE Command	Request Payload Type	Direction	Description
supply_opks	supply_opks_request_payload	Inbound	Supply 100 new opks to server.

E2EE Message Request

The e2ee_message payload in this request has three types specified by the msg_type attribute:

PRE_KEY:

When SKISSM is creating an outbound session, a PRE_KEY type e2ee_message will be sent to remote peer that provides necessary information for building a corresponding inbound session. An e2ee_message with PRE_KEY msg_type indicates that e2ee_pre_key_payload is the payload type provided in the message sending request.

MESSAGE:

When SKISSM uses one-to-one session to send message, a MESSAGE type e2ee_message will be sent to remote peer to remote peer.
An e2ee_message with MESSAGE msg_type indicates that e2ee_msg_payload is the payload type provided in the message sending request. After decrypt the ciphertext of e2ee_msg_payload, the decrypted data can be unpacked as an e2ee_plaintext message that has two types specified by e2ee_plaintext_type attribute:

COMMON_MSG:

The payload of e2ee_plaintext is the application message protected by SKISSM.

GROUP_PRE_KEY:

When SKISSM is creating an outbound group session, a GROUP_PRE_KEY type e2ee_msg_payload will be sent to remote peer. The payload of e2ee_plaintext can be unpacked as e2ee_group_pre_key_payload that provides necessary information for building a corresponding inbound group session.

GROUP_MESSAGE:

When user uses group session to send message, a GROUP_MESSAGE type e2ee_message will be sent to remote peer to remote peer. An e2ee_message with MESSAGE msg_type indicates that e2ee_group_msg_payload is the payload type provide

E2EE Command	Request Payload Type	Direction	Description
get_group	get_group_request_payload	outbound	Get the information of a group by providing group address.

d in the message sending request.

E2EE Command	Request Payload Type	Direction	Description
e2ee_msg	e2ee_message	Inbound outbound	Send e2ee message to remote peer, or receive e2ee message from remote peer.

Create Group Request

This request is used to create a new group. The group address is returned in create_group_response_payload. On the other hand, messaging server will forward this request to each group member. SKISSM will create an outbound group session on serving this request from server.

E2EE Command	Request Payload Type	Direction	Description
create_group	create_group_request_payload	Inbound outbound	Send this request to create a new group by providing group name, and group member address. If this is the case of inbound request, SKISSM is managed to automatically create a new outbound group session and send GROUP_PRE_KEY message to each group members.

Get Group Request

This request is served for getting information of a group. The group name and member addresses is returned in get_group_response_payload.

Add Group Members Request

This request is used to add some group members to an existing group. On the other hand, messaging server will forward this request to each pre-existed group member. SKISSM will create another new outbound group session on serving this request from server.

E2EE Command	Request Payload Type	Direction	Description
add_group_members	add_group_members_request_payload	inbound outbound	Send this request to add some new group members by providing new group member address. If this is the case of inbound request, SKISSM is managed to automatically create another new outbound group session and send GROUP_PRE_KEY message to each group members.

Remove Group Members Request

This request is used to remove some group members from an existing group. On the other hand, messaging server will forward this request to each post-existed group member. SKISSM will create another new outbound group session on serving this request from server.

E2EE Command	Request Payload Type	Direction	Description
remove_group_members	remove_group_members_request_payload	Inbound outbound	Send this request to remove some group members by providing old group member address. If this is the case of inbound request, SKISSM is managed to automatically create another new outbound group session and send GROUP_PRE_KEY message to each group members.

References

[1] Trevor Perrin (editor) "The XEdDSA and VXEdDSA Signature Schemes", Revision 1, 2016-10-20. https://signal.org/docs/specifications/xeddsa/

[2] Moxie Marlinspike, Trevor Perrin (editor) "The X3DH Key Agreement Protocol", Revision 1, 2016-11-04. https://signal.org/docs/specifications/x3dh/

[3] Moxie Marlinspike, Trevor Perrin (editor) "The Double Ratchet Algorithm", Revision 1, 2016-11-20. https://signal.org/docs/specifications/doubleratchet/

[4] Moxie Marlinspike, Trevor Perrin (editor) "The Sesame Algorithm: Session Management for Asynchronous Message Encryption", Revision 2, 2017-04-14. https://signal.org/docs/specifications/sesame/

[5] Proto3 Language Guide, https://developers.google.com/protocol-buffers/docs/proto3

[6] A. Langley, M. Hamburg, and S. Turner, "Elliptic Curves for Security.", Internet Engineering Task Force; RFC 7748 (Informational); IETF, Jan-2016. http://www.ietf.org/rfc/rfc7748.txt

[7] S. Josefsson and I. Liusvaara "Edwards-Curve Digital Signature Algorithm (Ed- DSA)", Internet Engineering Task Force; RFC 8032 (Informational); IETF, Jan- 2017. https://tools.ietf.org/html/rfc8032

[8] J. Salowey, A. Choudhury, and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", Internet Engineering Task Force; RFC 5288 (Standards Track); IETF, August 2008. https://www.ietf.org/rfc/rfc5288.txt

[9] H. Krawczyk and P. Eronen "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. https://tools.ietf.org/html/rfc5869

[10] A collection of implementations of curve25519, an elliptic curve Diffie Hellman primitive "curve25519-donna", https://github.com/agl/curve25519-donna/tree/master

[11] ARM mbed "mbed TLS", https://tls.mbed.org