XEP-0384: OMEMO Encryption

Andreas Straub
mailto:andy@strb.org
xmpp:andy@strb.org

Daniel Gultsch
mailto:daniel@gultsch.de
xmpp:daniel@gultsch.de

Tim Henkes
mailto:me@syndace.dev

Klaus Herberth
xmpp:klaus@jsxc.org

Paul Schaub
mailto:vanitasvitae@riseup.net
xmpp:vanitasvitae@jabberhead.tk

Marvin Wißfeld
mailto:xmpp@larma.de
xmpp:jabber@larma.de

2025-04-07
Version 0.9.0

Status	Type	Short Name
Experimental	Standards Track	OMEMO

This specification defines a protocol for end-to-end encryption in one-to-one chats, as well as group chats where each participant may have multiple clients per account.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

1.1 Motivation

There are two main end-to-end encryption schemes in common use in the XMPP ecosystem, Off-the-Record (OTR) messaging (Current Off-the-Record Messaging Usage (XEP-0364) [1]) and OpenPGP (Current Jabber OpenPGP Usage (XEP-0027) [2]). Older OTR versions have had significant usability drawbacks for inter-client mobility. As OTR sessions existed between exactly two clients, the chat history would not be synchronized across other clients of the involved parties. Furthermore, OTR chats were only possible if both participants were online at the same time, due to how the rolling key agreement scheme of OTR worked. Some of those problems have been addressed in OTRv4. OpenPGP, while not suffering from these mobility issues, does not provide any kind of forward secrecy and is vulnerable to replay attacks. Additionally, PGP over XMPP uses a custom wireformat which is defined by convention rather than standardization, and involves quite a bit of external complexity. The wire format issues were resolved with OpenPGP for XMPP (XEP-0373) [3].
This XEP defines a protocol that leverages the Double Ratchet encryption scheme to provide multi-end to multi-end encryption, allowing messages to be synchronized securely across multiple clients, even if some of them are offline. The Double Ratchet encryption scheme is based on work by Trevor Perrin and Moxie Marlinspike and was first published as the Axolotl protocol. The specification for the protocol is available in the public domain.

1.2 Overview

The general idea behind this protocol is to maintain separate, long-standing Double Ratchet-encrypted sessions with each device of each contact (as well as with each of our other devices), which are used as secure key transport channels. In this scheme, each message is encrypted with a fresh, randomly generated encryption key. An encrypted header is added to the message for each device that is supposed to receive it. These headers simply contain the key that the payload message is encrypted with, and they are separately encrypted using the session corresponding to the counterpart device. The encrypted payload is sent together with the headers as a <message> stanza. Individual recipient devices can decrypt the header item intended for them, and use the contained payload key to decrypt the payload message.
As the encrypted payload is common to all recipients, it only has to be included once, reducing overhead. Furthermore, the transparent handling by the Double Ratchet encryption scheme of messages that were lost or received out of order, as well as those sent while the recipient was offline, is maintained by this protocol. As a result, in combination with Message Carbons (XEP-0280) [4] and Message Archive Management (XEP-0313) [5], the desired property of inter-client history synchronization is achieved.

[1] XEP-0364: Current Off-the-Record Messaging Usage <https://xmpp.org/extensions/xep-0364.html>.
[2] XEP-0027: Current Jabber OpenPGP Usage <https://xmpp.org/extensions/xep-0027.html>.
[3] XEP-0373: OpenPGP for XMPP <https://xmpp.org/extensions/xep-0373.html>.
[4] XEP-0280: Message Carbons <https://xmpp.org/extensions/xep-0280.html>.
[5] XEP-0313: Message Archive Management <https://xmpp.org/extensions/xep-0313.html>.

While in the future a dedicated key server component could be used to distribute key material for session creation, the current specification relies on Publish-Subscribe (XEP-0060) [6] and Personal Eventing Protocol (XEP-0163) [7] to publish and acquire key bundles.

2 Requirements

It is a result of XMPP's federated nature that a message may pass more than just one server. Therefore it is in the users' interest to secure their communication from any intermediate host. End-to-end encryption is an efficient way to protect any data exchanged between sender and receiver against passive and active attackers such as servers and network nodes.
OMEMO is an end-to-end encryption protocol based on the Double Ratchet specified in section Double Ratchet. It provides the following guarantees under the threat model described in the next section:

- Confidentiality: Nobody else except sender and receiver is able to read the content of a message.
- Forward Secrecy: Compromised key material does not compromise previous message exchanges. It has been demonstrated, that OMEMO provides only weak forward secrecy (it protects the session key only once both parties complete the key exchange).
- Break-in Recovery: A session which has been compromised due to leakage of key material recovers from the compromise after a few communication rounds.
- Authentication: Every peer is able to authenticate the sender or receiver of a message, even if the details of the authentication process is out-of-scope for this specification.
- Integrity: Every peer can ensure that a message was not changed by any intermediate node.
- Deniability: X3DH is weakly offline deniable and provides no online deniability, as far as the research shows.
- Asynchronicity: The usability of the protocol does not depend on the online status of any participant.

OMEMO is not intended to protect against the following use cases:

- An attacker has permanent access to your device. In this case, the attacker may extract decrypted messages from the device, eg. from the applications database. If the access is temporary, security will eventually be restored through break-in recovery.

[6] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.
[7] XEP-0163: Personal Eventing Protocol <https://xmpp.org/extensions/xep-0163.html>.

- You lost your device and an attacker can read messages on your notification screen.
- Any kind of denial-of-service attack.
- tbc

Trust management is a difficult topic, which is out of scope of this document.

2.1 Threat Model

The use case for OMEMO is a situation where the content of a conversation needs to be protected, but where the servers the message passes by can't be trusted to keep the content of the message secret. For example when information that is under strict embargo needs to passed within an organization and the server administrator is not one of the persons cleared to see the information or when a couple is exchanging intimate messages and they want to avoid leaking of those messages to the server administrator.
The OMEMO protocol protects against passive and active attackers which are able to read, modify, replay, delay and delete messages. The OMEMO protocol does not protect against attackers who rely on metadata and traffic analysis. The quality of the verification of the conversation participants OMEMO identity keys determines the level of protection OMEMO offers.

3 Glossary

Device A communication end point, i.e. a specific client instance

OMEMO element An <encrypted> element in the urn:xmpp:omemo:2 namespace

Bundle A collection of publicly accessible data used by the X3DH key agreement protocol that can be used to build a session with a device, namely its public IdentityKey, a signed PreKey with corresponding signature, and a list of (single use) PreKeys.

rid The device id of the intended recipient of the containing <key>

sid The device id of the sender of the containing OMEMO element

4 Protocol Definition

4.1 Overview

This protocol uses the DoubleRatchet [8] encryption scheme in conjunction with the X3DH [9] key agreement protocol. The following section provides detailed technical information about the protocol that should be sufficient to build an implementation of the OMEMO Double Ratchet.

[8] The Double Ratchet Algorithm <https://signal.org/docs/specifications/doubleratchet/>.
[9] The X3DH Key Agreement Protocol <https://www.signal.org/docs/specifications/x3dh/>.

Readers who do not intend to build an OMEMO-compatible library can safely skip this section, relevant details are repeated where needed.

4.2 Key Exchange

The X3DH [10] key agreement protocol was specified by Trevor Perrin and Moxie Marlinspike and placed under the public domain. OMEMO uses a modified version of this key agreement protocol with the following parameters/settings:

curve Curve25519/Ed25519

hash function SHA-256

info string "OMEMO X3DH"

signed PreKey rotation period Signed PreKeys SHOULD be rotated periodically once a week to once a month. A faster or slower rotation period should not be required.

time to keep the private key of the old signed PreKey after rotating it The private key of the old signed PreKey SHOULD be kept for another rotation period as defined above, to account for delayed messages using the old signed PreKey.

number of PreKeys to provide in the bundle The bundle SHOULD always contain around 100 PreKeys.

minimum number of PreKeys to provide in the bundle The bundle MUST always contain at least 25 PreKeys.

usage of PeyKeys All key exchanges MUST use a PreKey, key exchanges that don't use a PreKey MUST be rejected.

associated data The associated data is created by concatenating the IdentityKeys of Alice and Bob: AD = Encode(IK_A) || Encode(IK_B). Alice is the party that actively initiated the key exchange, while Bob is the party that passively accepted the key exchange.

XEdDSA OMEMO does not mandate the usage of XEdDSA The XEdDSA and VXEdDSA Signature Schemes <https://www.signal.org/docs/specifications/xeddsa/>. with X3DH The X3DH Key Agreement Protocol <https://www.signal.org/docs/specifications/x3dh/>. for the IdentityKey. Instead, there are three simple rules that implementations MUST follow:

Implementations must use the birational map between the curves Curve25519 and Ed25519 to convert the public part of the IdentityKey whenever required, as defined in RFC 7748 RFC 7748: Elliptic Curves for Security <http://tools.ietf.org/html/rfc7748>. (on page 5). Implementations must be able to perform X25519 (ECDH on Curve25519) using

[10] The X3DH Key Agreement Protocol <https://www.signal.org/docs/specifications/x3dh/>.

the IdentityKey. Implementations must be able to create EdDSA-compatible signatures on the curve Ed25519 using the IdentityKey.

There are essentially two ways in which libraries can fulfill these requirements:

Libraries can use a Curve25519 key pair as their internal IdentityKey. In this case, the IdentityKey can be used for X25519 directly, and XEdDSA has to be used to produce EdDSA-compatible signatures. Note that libsignal by default does NOT use XEdDSA. libsignal includes XEdDSA though and has to be modified to use that to be compatible with OMEMO. Libraries can use an Ed25519 key pair as their internal IdentityKey. In this case, the IdentityKey can create EdDSA-compatible signatures directly, and has to be converted first to perform X25519.

Note that this decision is purely local to each client and OMEMO library. The public key is ALWAYS transferred in its Ed25519 form and only valid EdDSA signatures are transferred. The choice between Curve25519 and Ed25519 affects the definition of the Sig(PK, M) and DH(PK1, PK2) functions as defined below.

Sig(PK, M) If the IdentityKey pair is chosen to be a Curve25519 key pair, the definition of Sig(PK, M) found in the X3DH specification applies. If the IdentityKey pair is chosen to be an Ed25519 key pair, the following definition applies: Sig(PK, M) represents the byte sequence that is an Ed25519 signature on the byte sequence M and verifies with public key PK, and which was created by signing M with PK's corresponding private key. The byte-format of the signature is defined in RFC 8032 RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA) <http://tools.ietf.org/html/rfc8032>..

DH(PK1, PK2) The original definition of DH(PK1, PK2) found in the X3DH specification applies with one exception: if the IdentityKey pair is chosen to be an Ed25519 key pair and either PK1 or PK2 corresponds to the IdentityKey, the respective key first has to be converted into its Curve25519 equivalent (see above). This conversion is implemented for example by libsodium, which exports the conversion as crypto_sign_ed25519_sk_to_curve25519 and crypto_sign_ed25519_pk_to_curve25519 for the private and public key, respectively (documented on libsodium.org).

Encode(PK) The public part of the IdentityKey pair is encoded as defined in RFC 8032 RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA) <http://tools.ietf.org/html/rfc8032>.. Note that the IdentityKey is always transferred in its Ed25519 form. When using a Curve25519 key pair internally, the public key has to be converted to Ed25519 first. Curve25519 public keys are encoded using the little-endian encoding of the u-coordinate as specified in RFC 7748 RFC 7748: Elliptic Curves for Security <http://tools.ietf.org/html/rfc7748>..

The key exchange is done just-in-time when sending the first message to a device. Thus, each key exchange message always also contains encrypted content as produced by the Double Ratchet encryption scheme below.

4.3 Double Ratchet

NOTE: OMEMOMessage.proto, OMEMOAuthenticatedMessage.proto and OMEMOKeyExchange.proto refer to the protobuf structures as defined in the Protobuf Schemas.
The DoubleRatchet [11] encryption scheme was specified by Trevor Perrin and Moxie Marlinspike and placed under the public domain. OMEMO uses this protocol with the following parameters/settings:

ratchet initialization The Double Ratchet is initialized using the shared secret, ad and public keys as yielded by the X3DH key agreement protocol, as explained in the Double Ratchet specification. Additionally, the ephemeral key pair (ek) used by the X3DH key agreement is stored with the session.

MAX_SKIP Storing skipped message keys introduces two potential DoS attacks. First, the maximum number of skipped message keys to store has to be limited, otherwise an attacker could fill the storage of the receiving device with skipped message keys. It is RECOMMENDED to keep 1000 skipped message keys around per session. Second, the maximum number of skipped message keys in a single message has to be limited, otherwise, with just a single malicious message, an attacker could make the receiving device calculate up to around 2^32 skipped message keys. The choice of this limit depends on the hardware capabilities of the device, though modern hardware is safe to choose values of around 1000 here too.

deletion policy for skipped message keys As soon as the maximum number of skipped message keys are stored for a session, keys for that session are discarded on a FIFO basis to make space for new skipped message keys. Implementations SHOULD NOT keep skipped message keys around forever, but discard old keys on a different implementation-defined policy. It is RECOMMENDED to base this policy on deterministic events rather than time.

authentication tag truncation Authentication tags are truncated to 16 bytes/128 bits by cutting off excess bytes from the end.

CONCAT(ad, header) CONCAT(ad, header) = ad || OMEMOMessage.proto(header) NOTE: the OMEMOMessage.proto is initialized without the ciphertext, which is optional. NOTE: Implementations are not strictly required to return a parseable byte array, as the unpacked/parsed data is required later in the protocol.

KDF_RK(rk, dh_out) HKDF-SHA-256 using the root key (rk) as HKDF salt, the output of a Diffie-Hellman (dh_out) as HKDF input material and "OMEMO Root Chain" as HKDF info.

KDF_CK(ck) HMAC-SHA-256 using a chain key (ck) as the HMAC key, a single byte constant 0x01 as HMAC input to produce the next message key (mk) and a single byte constant 0x02 as HMAC input to produce the next chain key.

[11] The Double Ratchet Algorithm <https://signal.org/docs/specifications/doubleratchet/>.

ENCRYPT(mk, plaintext, associated_data) The encryption step uses authenticated encryption consisting of AES-256-CBC with HMAC-SHA-256.

Use HKDF-SHA-256 to generate 80 bytes of output from the message key by providing mk as HKDF input, 256 zero-bits as HKDF salt and "OMEMO Message Key Material" as HKDF info. Divide the HKDF output into a 32-byte encryption key, a 32-byte authentication key and a 16 byte IV. Encrypt the plaintext (which consists of a 32 bytes key and a 16 bytes HMAC as specified in the section about Message Encryption) using AES-256-CBC with PKCS#7 padding, using the encryption key and IV derived in the previous step. Split the associated data as returned by CONCAT into the original ad and the OMEMOMessage.proto structure. Add the ciphertext to the OMEMOMessage.proto structure. Serialize the OMEMOMessage.proto structure into a parseable byte array. To avoid potential problems regarding non-uniqueness of the serialization, make sure to only serialize once and to use that exact byte sequence in the following steps. Concatenate the ad and the OMEMOMessage.proto structure into a parseable byte array. The result builds the HMAC input material for the next step. Calculate the HMAC-SHA-256 using the authentication key and the input material as derived in the steps above. Truncate the output of the HMAC to 16 bytes/128 bits by cutting off excess bytes from the end. Put the serialized OMEMOMessage.proto structure and the HMAC into a new OMEMOAuthenticatedMessage.proto structure.

Information on the functions mentioned above can be found in the Double Ratchet specification.
If encrypting this message required a key exchange, the X3DH header data is placed into a new OMEMOKeyExchange.proto structure together with the OMEMOAuthenticatedMessage.proto structure.
To account for lost and out-of-order messages during the key exchange, OMEMOKeyExchange.proto structures are sent until a response by the recipient confirms that the key exchange was successfully completed. To do so, the X3DH header data is stored and added on each subsequent message until a response is received. This looks roughly as follows:

1. The first content is encrypted for a new recipient. This results in an X3DH header and a OMEMOAuthenticatedMessage.proto structure. Both are packed into an OMEMOKeyExchange.proto structure. The X3DH header is stored for following messages.

2. A second message is encrypted for the same recipient. This results in only an OMEMOAuthenticatedMessage.proto structure, as a new key exchange is not required. Together with the X3DH header that was stored in the previous step, an OMEMOKeyExchange.proto structure is constructed and sent to the recipient.

When receiving an OMEMOKeyExchange, the receiving device checks if it already has a Double Ratchet session with the sending device. If that is the case, the device compares the ephemeral public key stored in the Double Ratchet state with the ephemeral public key in the OMEMOKeyExchange.proto structure. If both are equal, the receiving device only processes

the OMEMOAuthenticatedMessage.proto contained in the OMEMOKeyExchange.proto. Otherwise, it processes the whole OMEMOKeyExchange.proto structure.

4.4 Message Encryption

The contents are encrypted and authenticated using a combination of AES-256-CBC and HMAC-SHA-256.

1. Generate 32 bytes of cryptographically secure random data, called key in the remainder of this algorithm.
2. Use HKDF-SHA-256 to generate 80 bytes of output from the key by providing the key as HKDF input, 256 zero-bits as HKDF salt and ”OMEMO Payload” as HKDF info.
3. Divide the HKDF output into a 32-byte encryption key, a 32-byte authentication key and a 16 byte IV.
4. Encrypt the plaintext using AES-256-CBC with PKCS#7 padding, using the encryption key and IV derived in the previous step.
5. Calculate the HMAC-SHA-256 using the authentication key and the ciphertext from the previous steps. Truncate the output of the HMAC to 16 bytes/128 bits by cutting off excess bytes from the end.
6. Concatenate the key and the HMAC, encrypt them using the Double Ratchet as specified above, once for each intended recipient. This yields one OMEMOKeyExchange or OMEMOAuthenticatedMessage per recipient device.

4.5 Message Decryption

The contents are decrypted by reversing the encryption steps.

1. Decrypt the key and HMAC from the OMEMOKeyExchange or OMEMOAuthenticatedMessage, encrypted using the Double Ratchet belonging to this device.
2. Use HKDF-SHA-256 to generate 80 bytes of output from the key by providing the key as HKDF input, 256 zero-bits as HKDF salt and ”OMEMO Payload” as HKDF info.
3. Divide the HKDF output into a 32-byte encryption key, a 32-byte authentication key and a 16 byte IV.
4. Verify the (truncated) HMAC-SHA-256 using the authentication key derived in the previous step and the ciphertext.
5. Decrypt the ciphertext using AES-256-CBC with PKCS#7 padding, using the encryption key and IV derived in the previous steps.

5 Use Cases

5.1 Setup

To participate in OMEMO-encrypted chats, clients need to set up an OMEMO library and generate a device id, which is a randomly generated integer between 1 and 2^31 - 1 (the positive numbers of a signed 32 bit integer, without 0). The device id must be unique for the account.

5.2 Discovering peer support

In order to determine whether a given contact has devices that support OMEMO, the devices node in PEP is consulted. Devices MUST subscribe to urn:xmpp:omemo:2:devices via PEP, so that they are informed whenever their contacts add a new device. They MUST cache the most up-to-date version of the device list.

Listing 1: Devicelist update received by subscribed clients

```
<message from='juliet@capulet.lit'
         to='romeo@montague.lit'
         type='headline'
         id='update_01'>
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='urn:xmpp:omemo:2:devices'>
      <item id='current'>
        <devices xmlns='urn:xmpp:omemo:2'>
          <device id='12345' />
          <device id='4223' label='Gajim on Ubuntu Linux' labelsig='
              b64/encoded/data' />
        </devices>
      </item>
    </items>
  </event>
</message>
```

Additionally, a strategy is required for dealing with OMEMO-encrypted chats between accounts without mutual presence subscription. Without presence, PEP can not be relied on to always have an up-to-date version of the device list cached. A few potential strategies for dealing with this issue are:

- Subscribe to the missing device list nodes directly using Publish-Subscribe (XEP-0060) [12] §6.1. This is a viable option in use-cases where potential chat partners are known, for example in IM-clients that keep message history and thus have a list of accounts with which OMEMO-encrypted messages have been exchanged before.

[12] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.

- Manually fetch the device list when a new encrypted message is likely to be sent soon, for example when the message input field receives focus or is being typed into.
- Manually fetch the device list right before sending an encrypted message. This strategy is not recommended for interactive clients where it may cause noticable delays.

5.3 Announcing support

5.3.1 Device list

In order for other devices to be able to initiate a session with a given device, it first has to announce itself by adding its device id to the devices PEP node.
It is REQUIRED to set the access model of the urn:xmpp:omemo:2:devices node to 'open' to give entities without presence subscription read access to the devices and allow them to establish an OMEMO session. Not having presence subscription is a common occurrence on the first few messages between two contacts and can also happen fairly frequently in group chats as not every participant had prior communication with every other participant.
The access model can be changed efficiently by using publish-options.
The device element MAY contain an attribute called label, which is a user defined string describing the device that published that bundle. It is RECOMMENDED to keep the length of the label under 53 Unicode code points. If a label is set, it MUST be accompanied by an attribute called labelsig, which contains a signature of the label to prevent malicious insertion or replacement of labels. The value of the labelsig attribute is found by encoding the label to bytes using utf-8, signing the label using the identity key (refer to Sig(PK, M) in Key Exchange), and encoding the signature to text using base64. The label attribute MUST be ignored if no labelsig attribute is present or the signature does not pass validation.

Listing 2: Adding the own device id to the list

```
<iq from='juliet@capulet.lit' type='set' id='announce1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='urn:xmpp:omemo:2:devices'>
      <item id='current'>
        <devices xmlns='urn:xmpp:omemo:2'>
          <device id='12345' label='Dino␣on␣Lenovo␣Thinkpad␣T495'
              labelsig='b64/encoded/data' />
          <device id='4223' />
          <device id='31415' label='Conversations␣on␣Pixel␣3' labelsig
              ='b64/encoded/data' />
        </devices>
      </item>
    </publish>
    <publish-options>
      <x xmlns='jabber:x:data' type='submit'>
        <field var='FORM_TYPE' type='hidden'>
          <value>http://jabber.org/protocol/pubsub#publish-options</
              value>
```

```
        </field>
        <field var='pubsub#access_model'>
          <value>open</value>
        </field>
      </x>
    </publish-options>
  </pubsub>
</iq>
```

NOTE: as per Publish-Subscribe (XEP-0060) [13] §12.20, it is RECOMMENDED for the publisher to specify an ItemID of "current" to ensure that the publication of a new item will overwrite the existing item.
This step presents the risk of introducing a race condition: Two devices might simultaneously try to announce themselves, unaware of the other's existence. The second device would overwrite the first one. To mitigate this, devices MUST check that their own device id is contained in the list whenever they receive a PEP update from their own account. If they have been removed, they MUST reannounce themselves.

5.3.2 Bundles

Furthermore, a device MUST publish its IdentityKey, a signed PreKey, and a list of PreKeys. This tuple is called a bundle and is provided by OMEMO libraries. Bundles are maintained as multiple items in a PEP node called urn:xmpp:omemo:2:bundles. Each bundle MUST be stored in a seperate item. The item id MUST be set to the device id.
A bundle is an element called 'bundle' in the urn:xmpp:omemo:2 namespace. It has a child element called 'spk' that contains the public part of the signed PreKey as base64 encoded data, a child element called 'spks' that contains the signed PreKey signature as base64 encoded data and a child element called 'ik' that contains the public part of the IdentityKey as base64 encoded data. PreKeys are multiple elements called 'pk' that each contain the public part of one PreKey as base64 encoded data. PreKeys are wrapped in an element called 'prekeys' which is a child of the bundle element. The 'spk' and the 'pk's are tagged with an 'id'-attribute which is a positive integer between 1 and 2^31 - 1 (the positive numbers of a signed 32 bit integer, without 0) that uniquely identifies the keys. The 'spk' and the 'pk's are considered separate, which means that an 'spk' can have the same 'id' as a 'pk'. These ids are used to save bandwidth during key exchanges, which refer to the keys using their id instead of their full public parts.
When publishing bundles a client MUST make sure that the urn:xmpp:omemo:2:bundles node is configured to store multiple items. This is not the default with Personal Eventing Protocol (XEP-0163) [14]. If the node doesn't exist yet it can be configured on the fly by using publish-options as described in Publish-Subscribe (XEP-0060) [15] §7.1.5. The value for 'pubsub#max_items' in publish_options MUST be set to 'max'. If the node did exist and was

[13] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.
[14] XEP-0163: Personal Eventing Protocol <https://xmpp.org/extensions/xep-0163.html>.
[15] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.

configured differently the bundle publication will fail. Clients MUST then reconfigure the node as described in XEP-0060 §8.2.

Listing 3: Publishing bundle information

```
<iq from='juliet@capulet.lit' type='set' id='annouce2'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='urn:xmpp:omemo:2:bundles'>
      <item id='31415'>
        <bundle xmlns='urn:xmpp:omemo:2'>
          <spk id='0'>b64/encoded/data</spk>
          <spks>b64/encoded/data</spks>
          <ik>b64/encoded/data</ik>
          <prekeys>
            <pk id='0'>b64/encoded/data</pk>
            <pk id='1'>b64/encoded/data</pk>
            <!-{}- … -{}->
            <pk id='99'>b64/encoded/data</pk>
          </prekeys>
        </bundle>
      </item>
    </publish>
    <publish-options>
      <x xmlns='jabber:x:data' type='submit'>
        <field var='FORM_TYPE' type='hidden'>
          <value>http://jabber.org/protocol/pubsub#publish-options</
              value>
        </field>
        <field var='pubsub#max_items'>
          <value>max</value>
        </field>
      </x>
    </publish-options>
  </pubsub>
</iq>
```

As with the urn:xmpp:omemo:2:devices node it is REQUIRED to set the access model of the urn:xmpp:omemo:2:bundles to open.
The access model can be changed efficiently by using publish-options.

Listing 4: Publishing bundle information with an open access model

```
<iq from='juliet@capulet.lit' type='set' id='annouce2'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='urn:xmpp:omemo:2:bundles'>
      <item id='31415'>
        <bundle xmlns='urn:xmpp:omemo:2'>
          <!-{}- … -{}->
        </bundle>
```

```
      </item>
    </publish>
    <publish-options>
      <x xmlns='jabber:x:data' type='submit'>
        <field var='FORM_TYPE' type='hidden'>
          <value>http://jabber.org/protocol/pubsub#publish-options</
              value>
        </field>
        <field var='pubsub#max_items'>
          <value>max</value>
        </field>
        <field var='pubsub#access_model'>
          <value>open</value>
        </field>
      </x>
    </publish-options>
  </pubsub>
</iq>
```

5.4 Building a session

In order to build a session with a device, their bundle information is fetched.

Listing 5: Fetching a device's bundle information

```
<iq type='get'
    from='romeo@montague.lit'
    to='juliet@capulet.lit'
    id='fetch1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='urn:xmpp:omemo:2:bundles'>
      <item id='31415'/>
    <items>
  </pubsub>
</iq>
```

A random pk entry is selected, and used to build an OMEMO session.

5.5 Sending a message

In order to send a message, extension elements that are deemed sensible first have to be encrypted. For this purpose, extensions that are only intended to be accessible to the recipient are placed inside a Stanza Content Encryption (XEP-0420) [16] <envelope/> element, which is then encrypted using a message key. For this reason OMEMO defines its own SCE profile.

[16] XEP-0420: Stanza Content Encryption <https://xmpp.org/extensions/xep-0420.html>.

5.5.1 SCE Profile

An OMEMO SCE <envelope/> element

- MUST contain an <rpad/> affix element. This is used to prevent an attacker from gaining insights about the content of a message based on the length of the ciphertext.
- MAY contain a <time/> affix element. This can be used to prevent the server from modifying the order in which messages from different sending devices have been sent.
- SHOULD contain a <from/> affix element.
- MUST contain a <to/> affix element whenever a message is sent via a group chat (MUC/MIX). This is used to prevent the server from silently converting a group message into a private message and vice versa.

Listing 6: Plaintext SCE envelope element

```
<envelope xmlns='urn:xmpp:sce:1'>
  <content>
    <body xmlns='jabber:client'>
     Hello World!
    </body>
  </content>
  <rpad>...</rpad>
  <from jid='romeo@montague.lit'/>
</envelope>
```

5.5.2 Encryption

The <envelope/> element is encrypted as described in the section about Message Encryption. Clients MUST only consider the devices on the urn:xmpp:omemo:2:devices node of each recipient (i.e. including their own devices node, but excluding itself).

5.5.3 Message structure description

An OMEMO encrypted message is specified to include an <encrypted> element in the urn:xmpp:omemo:2 namespace. It contains up to two child nodes, the <header> and the <payload/> element. The <header> element must always be present, the <payload/> element must be present unless an empty OMEMO message is sent, as described below. The <header> element has an attribute named 'sid' referencing the device id of the sending device and contains one or multiple <keys> elements, each with an attribute 'jid' of one of the recipients bare JIDs, as well as one or multiple <key> elements. A <key> element has an attribute named 'rid' referencing the device id of the recipient device, and an attribute named 'kex' which defaults to 'false' and indicates if the enclosed encrypted message includes a key exchange.

The key and HMAC encrypted using the long-standing OMEMO session for that recipient device are encoded using base64 and placed as text content into the <key> element. The encrypted <envelope/> element is encoded using base64 and placed as text content into the <payload/> element.
A special case are *empty* OMEMO messages, which are used in various places throughout the protocol purely to manage sessions and not to transfer content. With empty OMEMO messages, the step of creating and encrypting the <payload/> element is skipped. Instead of encrypting the key and authentication tag of the <payload/> ciphertext with the Double Ratchet session, 32 zero-bytes are encrypted with the Double Ratchet session directly. The resulting OMEMOKeyExchange or OMEMOAuthenticatedMessage are put into <key> elements as usual, but the <payload/> element is omitted altogether, so that the <encrypted> element only contains a <header>.

Listing 7: Sending a message

```
<message to='juliet@capulet.lit' from='romeo@montague.lit' id='send1'>
  <encrypted xmlns='urn:xmpp:omemo:2'>
    <header sid='27183'>
      <keys jid='juliet@capulet.lit'>
        <key rid='31415'>b64/encoded/data</key>
      </keys>
      <keys jid='romeo@montague.lit'>
        <key rid='1337'>b64/encoded/data</key>
        <key kex='true' rid='12321'>b64/encoded/data</key>
        <!-{}- ... -{}->
      </keys>
    </header>
    <payload>
      base64/encoded/message/key/encrypted/envelope/element
    </payload>
  </encrypted>
  <store xmlns='urn:xmpp:hints'/>
</message>
```

5.6 Receiving a message

When an OMEMO element is received, the client MUST check whether there is a <keys> element with a jid attribute matching its own bare jid and an inner <key> element with a rid attribute matching its own device id. If this is not the case the message was not encrypted for this particular device and a warning message SHOULD be displayed instead. If such an element exists, the client checks whether the element's contents are an OMEMOKeyExchange. If this is the case, a new session is built from this received element. The client MUST then republish their bundle information, replacing the used PreKey, such that it won't be used again by a different client. If the client already has a session with the sender's device, it MUST replace this session with the newly built session. The client MUST eventually delete the private key belonging to the PreKey after use (this is subject to the Business rules).

If the element's contents are an OMEMOAuthenticatedMessage, and the client has a session with the sender's device, it tries to decrypt the OMEMOAuthenticatedMessage using this session. If the decryption fails or there is no session with the sending device, a warning message SHOULD be displayed instead. Also refer to the section about recovering from broken sessions in the Business Rules.
After either the OMEMOKeyExchange or the OMEMOAuthenticatedMessage is decrypted, the content is decrypted as described in the section about Message Decryption.

5.7 Opt-out

An account can signal to a peer that it wants to stop communicating using OMEMO encrypted messages and would like to proceed in plain text instead. To do that any of that account's devices sends an <opt-out/> element qualified by the urn:xmpp:omemo:2 namespace to all intended recipient devices inside an encrypted stanza. The element MAY contain a child element <reason>. If a device is receiving an encrypted stanza containing an <opt-out/> element, it SHOULD display the information, that the peer would like to receive plain text messages. To prevent that the user is accidentally sending plaintext messages, the client MUST block all outgoing message until the user has confirmed the switch to plaintext. Any existing double ratchet sessions SHOULD remain intact. At any point any party MAY revert their decision and go back to sending OMEMO encrypted messages again.

Listing 8: A client signaling that it's account no longer wants to receive OMEMO-encrypted messages

```
<envelope xmlns='urn:xmpp:sce:1'>
  <content>
    <opt-out xmlns='urn:xmpp:omemo:2'>
      <reason>
        Sorry, but for compliance reasons I need a permanent,
        server-side, record of our conversation.
      </reason>
    </opt-out>
  </content>
</envelope>
```

5.8 Group Chats

NOTE: OMEMO encrypted group chats are currently specified to work with Multi-User Chat (XEP-0045) [17]. This XEP might be updated in the future to specify the usage of OMEMO in conjunction with Mediated Information eXchange (MIX) (XEP-0369) [18].
A Multi-User Chat room that supports OMEMO MUST be configured non-anonymous and SHOULD be configured members-only.

[17] XEP-0045: Multi-User Chat <https://xmpp.org/extensions/xep-0045.html>.
[18] XEP-0369: Mediated Information eXchange (MIX) <https://xmpp.org/extensions/xep-0369.html>.

A participant wanting to send a message to a group chat MUST first retrieve the members list and then fetch the device list for each member (via pubsub and to their real JIDs) and then subsequently fetch all bundles referenced by the device lists.

5.8.1 Retrieving and maintaining members list

On join a participant MUST request the member list, the admin list and the owner list as described in Multi-User Chat (XEP-0045) [19] §9.5, XEP-0045 §10.8, and XEP-0045 §10.5 respectively. The real JIDs from those three lists MUST be combined as the recipients of OMEMO encrypted messages. This includes recipients who are currently offline. Once joined a participant MUST keep track of affiliation changes that occur in the room. This is both for removals (users getting banned or have their affiliation set to none) and users becoming members, admins or owners.

5.8.2 Fetching devices and bundles

Before sending a message a participant MUST explicitly fetch device lists (if not already cached) for each of the members.

Listing 9: Juliet fetching devices for Romeo and Mercutio

```
<iq type='get' from='juliet@capulet.lit' to='romeo@montague.lit' id='
    gfetch0'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='urn:xmpp:omemo:2:devices'/>
  </pubsub>
</iq>
<iq type='get' from='juliet@capulet.lit' to='mercutio@verona.lit' id='
    gfetch1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='urn:xmpp:omemo:2:devices'/>
  </pubsub>
</iq>
```

Listing 10: Juliet fetches bundles for Romeo and Mercutio

```
<iq type='get' from='juliet@capulet.lit' to='romeo@montague.lit' id='
    gfetch2'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='urn:xmpp:omemo:2:bundles'>
      <item id='123'/>
    <items>
  </pubsub>
</iq>
```

[19] XEP-0045: Multi-User Chat <https://xmpp.org/extensions/xep-0045.html>.

```
<iq type='get' from='juliet@capulet.lit' to='mercutio@verona.lit' id='
    gfetch3'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <items node='urn:xmpp:omemo:2:bundles'>
      <item id='456'/>
    <items>
  </pubsub>
</iq>
```

5.8.3 Sending a message

Sending a message to a group chat is similiar to sending a message in a 1:1 conversation. Instead of the <header> element having two <keys> elements (one for the recipient and one for other devices of the sender) it will contain multiple <keys> elements. One for each participant of the room; including, again, other devices of the sender.

Listing 11: Juliet sends a message to a group chat with Romeo and Mercutio

```
<message
    from='juliet@capulet.lit/balcony'
    to='secret-room@conference.capulet.lit'
    type='groupchat'>
  <encrypted xmlns='urn:xmpp:omemo:2'>
    <header sid='27183'>
      <keys jid='juliet@capulet.lit'>
        <key rid='31415'>b64/encoded/data</key>
      </keys>
      <keys jid='romeo@montague.lit'>
        <key kex='true' rid='123'>b64/encoded/data</key>
      </keys>
      <keys jid='mercutio@verona.lit'>
        <key kex='true' rid='456'>b64/encoded/data</key>
      </keys>
    </header>
    <payload>
      base64/encoded/message/key/encrypted/envelope/element
    </payload>
  </encrypted>
  <store xmlns='urn:xmpp:hints'/>
</message>
```

6 Business Rules

Before publishing a freshly generated device id for the first time, a device MUST check whether that device id already exists, and if so, generate a new one.
Clients SHOULD NOT immediately fetch the bundle and build a session as soon as a new device

is announced. Before the first message is exchanged, the contact does not know which PreKey has been used (or, in fact, that any PreKey was used at all). As they have not had a chance to remove the used PreKey from their bundle announcement, this could lead to collisions where both Alice and Bob pick the same PreKey to build a session with a specific device. As each PreKey SHOULD only be used once, the party that sends their initial OMEMOKeyExchange later loses this race condition. This means that they think they have a valid session with the contact, when in reality their messages MAY be ignored by the other end. By postponing building sessions, the chance of such issues occurring can be drastically reduced. It is RECOMMENDED to construct sessions only immediately before sending a message.
After receiving an OMEMOKeyExchange and successfully building a new session, the receiving device SHOULD automatically respond with an empty OMEMO message (as per Sending a message) to the source of the OMEMOKeyExchange. This is to notify the device that the session initiation was completed successfully and that the device can stop sending OMEMOKeyExchanges.
When receiving a message that is not an OMEMOKeyExchange from a device there is no session with, clients SHOULD create a session with that device and notify it about the new session by responding with an empty OMEMO message as per Sending a message.
There are various reasons why decryption of an OMEMOKeyExchange or an OMEMOAuthenticatedMessage could fail. One reason is if the message was received twice and already decrypted once, in this case the client MUST ignore the decryption failure and not show any warnings/errors. In all other cases of decryption failure, clients SHOULD notify their users (if applicable), so that the users know they potentially missed a message.
If an OMEMOKeyExchange is received as part of a message catch-up mechanism (like Message Archive Management (XEP-0313) [20]) and used to establish a new session with the sender, the client SHOULD postpone deletion of the private key corresponding to the used PreKey until after the catch-up is completed. If this is done, the client MUST send an OMEMO encrypted message with empty SCE payload right after the key exchange is completed, to forward the ratchet and to move away from the possibly double-used PreKey. This practice can mitigate the previously mentioned race condition by preventing message loss.
OMEMO's forward secrecy and backup/restore mechanisms don't play well together. Restoring old data can lead to desynchronized, "broken" sessions. Because these cases exist, clients MUST offer a way to manually replace broken sessions. It is advisable to have a session replacement option per recipient/per chat, if applicable. Otherwise, at least an application-global session reset MUST be available.
When a client receives the first message for a given ratchet key with a counter of 53 or higher, it MUST send a heartbeat message. Heartbeat messages are empty OMEMO messages as per Sending a message. These heartbeat messages cause the ratchet to forward, thus consequent messages will have the counter restarted from 0.
When a client receives a message from a device id that is not on the device list, it SHOULD try to retrieve that user's devices node directly to ensure their local cached version of the devices list is up-to-date.
When the user of a client deactivates OMEMO for an account or globally, the client SHOULD

[20] XEP-0313: Message Archive Management <https://xmpp.org/extensions/xep-0313.html>.

delete the corresponding bundles and device ids from the PEP nodes. That way other clients should stop encrypting for that account.

7 Implementation Notes

7.1 Server side requirements

While OMEMO uses a Pubsub Service (Publish-Subscribe (XEP-0060) [21]) on the user's account it has more requirments than those defined in Personal Eventing Protocol (XEP-0163) [22]. The requirements are:

- The pubsub service MUST persist node items, i.e., the feature 'persistent-items' is announced by the service.
- The pubsub service MUST support publishing options as defined in Publish-Subscribe (XEP-0060) [23] §7.1.5.
- The pubsub service MUST support 'max' as a value for the 'pubsub#persist_items' node configuration.
- The pubsub service MUST support the 'open' access model for node configuration and 'pubsub#access_model' as a publish option.

8 Security Considerations

Clients MUST NOT use a newly built session (or any other session) to transmit data without ensuring trust first. If a client were to opportunistically start using sessions for sending without asking the user whether to trust a device first, an attacker could publish a fake device for this user, which would then receive copies of all messages sent by/to this user. A client MAY use such "not (yet) trusted" sessions for decryption of received messages, but in that case it SHOULD indicate the untrusted nature of such messages to the user. This rule does not apply to empty OMEMO messages (as per Sending a message) that are used purely to transfer key material, e.g. as part of heartbeat messages or automatic key exchange completion.
When prompting the user for a trust decision regarding a key, the client SHOULD present the user with a fingerprint in the form of a hex-string, QR code, or other unique representation, such that it can be compared by the user. To ensure interoperability between clients and older versions of OMEMO, the fingerprint SHOULD be chosen to be the public part of the IdentityKey in its byte-encoded Curve25519 form (see the notes on XEdDSA and the byte-encoding of public keys in the X3DH protocol section for details). When displaying the fingerprint as a hex-string, the RECOMMENDED way to make it easier to compare the fingerprint is to split

[21] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.
[22] XEP-0163: Personal Eventing Protocol <https://xmpp.org/extensions/xep-0163.html>.
[23] XEP-0060: Publish-Subscribe <https://xmpp.org/extensions/xep-0060.html>.

the lowercase hex-string into 8 substrings of 8 chars each, then coloring each group of 8 lowercase hex chars using Consistent Color Generation (XEP-0392) [24].
Clients MUST NOT react to decryption errors by initiating new sessions automatically and without user interaction.
While it is RECOMMENDED that clients postpone private key deletion until after message catch-up, the X3DH standard mandates that clients should not use duplicate-PreKey sessions for sending, so clients MAY delete such keys immediately for security reasons. For additional information on potential security impacts of this decision, refer to [25].

9 IANA Considerations

This document requires no interaction with the Internet Assigned Numbers Authority (IANA).

10 XMPP Registrar Considerations

10.1 Protocol Namespaces

This specification defines the following XMPP namespaces:

- urn:xmpp:omemo:2

10.2 Protocol Versioning

If the protocol defined in this specification undergoes a revision that is not fully backwards-compatible with an older version, the XMPP Registrar shall increment the protocol version number found at the end of the XML namespaces defined herein, as described in Section 4 of XEP-0053.

11 XML Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
           targetNamespace='urn:xmpp:omemo:2'
           xmlns='urn:xmpp:omemo:2'>

    <xs:element name='encrypted'>
        <xs:complexType>
```

[24] XEP-0392: Consistent Color Generation <https://xmpp.org/extensions/xep-0392.html>.
[25] Menezes, Alfred, and Berkant Ustaoglu. "On reusing ephemeral keys in Diffie-Hellman key agreement protocols." International Journal of Applied Cryptography 2, no. 2 (2010): 154-158.

```
            <xs:all>
                <xs:element ref='header'/>
                <xs:element ref='payload' minOccurs='0' maxOccurs='1'/
                    >
            </xs:all>
        </xs:complexType>
    </xs:element>

    <xs:element name='payload' type='xs:base64Binary'/>

    <xs:element name='header'>
        <xs:complexType>
            <xs:sequence maxOccurs='unbounded'>
                <xs:element ref='keys'/>
            </xs:sequence>
            <xs:attribute name='sid' type='xs:unsignedInt'/>
        </xs:complexType>
    </xs:element>

    <xs:element name='keys'>
        <xs:complexType>
            <xs:sequence maxOccurs='unbounded'>
                <xs:element ref='key'/>
            </xs:sequence>
            <xs:attribute name='jid' type='xs:string' use='required'/>
        </xs:complexType>
    </xs:element>

    <xs:element name='key'>
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base='xs:base64Binary'>
                    <xs:attribute name='rid' type='xs:unsignedInt' use
                        ='required'/>
                    <xs:attribute name='kex' type='xs:boolean' default
                        ='false'/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>

    <xs:element name='devices'>
        <xs:complexType>
            <xs:sequence minOccurs='0' maxOccurs='unbounded'>
                <xs:element ref='device'/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

```
<xs:element name='device'>
    <xs:complexType>
        <xs:attribute name='id' type='xs:unsignedInt' use='
            required'/>
        <xs:attribute name='label' type='xs:string'/>
        <xs:attribute name='labelsig' type='xs:base64Binary'/>
    </xs:complexType>
</xs:element>

<xs:element name='bundle'>
    <xs:complexType>
        <xs:all>
            <xs:element ref='spk'/>
            <xs:element ref='spks'/>
            <xs:element ref='ik'/>
            <xs:element ref='prekeys'/>
        </xs:all>
    </xs:complexType>
</xs:element>

<xs:element name='spk'>
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base='xs:base64Binary'>
                <xs:attribute name='id' type='xs:unsignedInt' use=
                    'required'/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:element>

<xs:element name='spks' type='xs:base64Binary'/>
<xs:element name='ik' type='xs:base64Binary'/>

<xs:element name='prekeys'>
    <xs:complexType>
        <xs:sequence maxOccurs='unbounded'>
            <xs:element ref='pk'/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name='pk'>
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base='xs:base64Binary'>
                <xs:attribute name='id' type='xs:unsignedInt' use=
                    'required'/>
            </xs:extension>
```

```
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

12 Protobuf Schema

```
message OMEMOMessage {
    required uint32 n          = 1;
    required uint32 pn         = 2;
    required bytes  dh_pub     = 3;
    optional bytes  ciphertext = 4;
}

message OMEMOAuthenticatedMessage {
    required bytes mac     = 1;
    required bytes message = 2; // Byte-encoding of an OMEMOMessage
}

message OMEMOKeyExchange {
    required uint32 pk_id  = 1;
    required uint32 spk_id = 2;
    required bytes  ik     = 3;
    required bytes  ek     = 4;
    required OMEMOAuthenticatedMessage message = 5;
}
```

13 Acknowledgements

Big thanks to Daniel Gultsch for mentoring me during the development of this protocol. Thanks to Thijs Alkemade and Cornelius Aschermann for talking through some of the finer points of the protocol with me. And lastly I would also like to thank Sam Whited, Holger Weiss, and Florian Schmaus for their input on the standard.
The authors would like to thank the Chaosdorf for hosting them during the development of version 0.4.0 of this specification.
Furthermore, the authors want to thank Sofía Celi for her feedback and input on the document.