XEP-0384: OMEMO Encryption

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

2017-06-02
Version 0.2

Status	Type	Short Name
Experimental	Standards Track	OMEMO

This specification defines a protocol for end-to-end encryption in one-on-one chats that 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]). OTR has significant usability drawbacks for inter-client mobility. As OTR sessions exist between exactly two clients, the chat history will not be synchronized across other clients of the involved parties. Furthermore, OTR chats are only possible if both participants are currently online, due to how the rolling key agreement scheme of OTR works. 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.
This XEP defines a protocol that leverages the SignalProtocol encryption 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 SignalProtocol is a cryptographic double ratched protocol based on work by Trevor Perrin and Moxie Marlinspike first published as the Axolotl protocol. While the protocol itself has specifications in the public domain, the protobuf-based wire format of the signal protocol is not fully documented. The signal protocol currently only exists in GPLv3-licensed implementations maintained by OpenWhisperSystems.

1.2 Overview

The general idea behind this protocol is to maintain separate, long-standing SignalProtocol-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 seperately 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, SignalProtocols's transparent handling 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) [3] and Message Archive Management (XEP-0313) [4], the desired property of inter-client history synchronization is achieved.
OMEMO currently uses version 3 SignalProtocol. Instead of a Signal key server, Personal

[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-0280: Message Carbons <https://xmpp.org/extensions/xep-0280.html>.
[4] XEP-0313: Message Archive Management <https://xmpp.org/extensions/xep-0313.html>.

Eventing Protocol (XEP-0163) [5] (PEP) is used to publish key data.

2 Requirements

- Provide forward secrecy
- Ensure chat messages can be deciphered by all (capable) clients of both parties
- Be usable regardless of the participants' online statuses
- Provide a method to exchange auxilliary keying material. This could for example be used to secure encrypted file transfers.

3 Glossary

3.1 General Terms

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

OMEMO element An <encrypted> element in the eu.siacs.conversations.axolotl namespace. Can be either MessageElement or a KeyTransportElement

MessageElement An OMEMO element that contains a chat message. Its <payload>, when decrypted, corresponds to a <message>'s <body>.

KeyTransportElement An OMEMO element that does not have a <payload>. It contains a fresh encryption key, which can be used for purposes external to this XEP.

Bundle A collection of publicly accessible data 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

3.2 SignalProtocol-specific

IdentityKey Per-device public/private key pair used to authenticate communications

PreKey A Diffie-Hellman public key, published in bulk and ahead of time

PreKeySignalMessage An encrypted message that includes the initial key exchange. This is used to transparently build sessions with the first exchanged message.

SignalMessage An encrypted message

[5] XEP-0163: Personal Eventing Protocol <https://xmpp.org/extensions/xep-0163.html>.

4 Use Cases

4.1 Setup

The first thing that needs to happen if a client wants to start using OMEMO is they need to generate an IdentityKey and a Device ID. The IdentityKey is a Curve25519 [6] public/private Key pair. The Device ID is a randomly generated integer between 1 and 2^31 - 1.

4.2 Discovering peer support

In order to determine whether a given contact has devices that support OMEMO, the devicelist node in PEP is consulted. Devices MUST subscribe to 'eu.siacs.conversations.axolotl.devicelist' 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 devicelist.

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='eu.siacs.conversations.axolotl.devicelist'>
      <item>
        <list xmlns='eu.siacs.conversations.axolotl'>
          <device id='12345' />
          <device id='4223' />
        </list>
      </item>
    </items>
  </event>
</message>
```

4.3 Announcing support

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 devicelist PEP node.

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='eu.siacs.conversations.axolotl.devicelist'>
      <item>
```

[6] Curve25519: new Diffie-Hellman speed records <http://cr.yp.to/ecdh/curve25519-20060209.pdf>.

```
        <list xmlns='eu.siacs.conversations.axolotl'>
          <device id='12345' />
          <device id='4223' />
          <device id='31415' />
        </list>
      </item>
    </publish>
  </pubsub>
</iq>
```

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.
Furthermore, a device MUST announce it's IdentityKey, a signed PreKey, and a list of PreKeys in a separate, per-device PEP node. The list SHOULD contain 100 PreKeys, but MUST contain no less than 20.

Listing 3: Announcing bundle information

```
<iq from='juliet@capulet.lit' type='set' id='announce2'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='eu.siacs.conversations.axolotl.bundles:31415'>
      <item>
        <bundle xmlns='eu.siacs.conversations.axolotl'>
          <signedPreKeyPublic signedPreKeyId='1'>
            BASE64ENCODED...
          </signedPreKeyPublic>
          <signedPreKeySignature>
            BASE64ENCODED...
          </signedPreKeySignature>
          <identityKey>
            BASE64ENCODED...
          </identityKey>
          <prekeys>
            <preKeyPublic preKeyId='1'>
              BASE64ENCODED...
            </preKeyPublic>
            <preKeyPublic preKeyId='2'>
              BASE64ENCODED...
            </preKeyPublic>
            <preKeyPublic preKeyId='3'>
              BASE64ENCODED...
            </preKeyPublic>
            <!-{}- ... -{}->
          </prekeys>
        </bundle>
```

```
      </item>
    </publish>
  </pubsub>
</iq>
```

4.4 Building a session

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

Listing 4: 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='eu.siacs.conversations.axolotl.bundles:31415'/>
  </pubsub>
</iq>
```

A random preKeyPublic entry is selected, and used to build a SignalProtocol session.

4.5 Sending a message

In order to send a chat message, its <body> first has to be encrypted. The client MUST use fresh, randomly generated key/IV pairs with AES-128 in Galois/Counter Mode (GCM). The 16 bytes key and the GCM authentication tag (The tag SHOULD have at least 128 bit) are concatenated and for each intended recipient device, i.e. both own devices as well as devices associated with the contact, the result of this concatenation is encrypted using the corresponding long-standing SignalProtocol session. Each encrypted payload key/authentication tag tuple is tagged with the recipient device's ID. The key element MUST be tagged with a prekey attribute set to true if a PreKeySignalMessage is being used. This is all serialized into a MessageElement, which is transmitted in a <message> as follows:

Listing 5: Sending a message

```
<message to='juliet@capulet.lit' from='romeo@montague.lit' id='send1'>
  <encrypted xmlns='eu.siacs.conversations.axolotl'>
    <header sid='27183'>
      <key rid='31415'>BASE64ENCODED...</key>
      <key prekey="true" rid='12321'>BASE64ENCODED...</key>
      <!-{}- ... -{}->
      <iv>BASE64ENCODED...</iv>
    </header>
    <payload>BASE64ENCODED</payload>
```

```
  </encrypted>
  <store xmlns='urn:xmpp:hints'/>
</message>
```

4.6 Sending a key

The client may wish to transmit keying material to the contact. This first has to be generated. The client MUST generate a fresh, randomly generated key/IV pair. The 16 bytes key and the GCM authentication tag (The tag SHOULD have at least 128 bit) are concatenated and for each intended recipient device, i.e. both own devices as well as devices associated with the contact, this key is encrypted using the corresponding long-standing SignalProtocol session. Each encrypted payload key/authentication tag tuple is tagged with the recipient device's ID. The key element MUST be tagged with a prekey attribute set to true if a PreKeySignalMessage is being used This is all serialized into a KeyTransportElement, omitting the <payload> as follows:

Listing 6: Sending a key

```
<encrypted xmlns='eu.siacs.conversations.axolotl'>
  <header sid='27183'>
    <key rid='31415'>BASE64ENCODED...</key>
    <key prekey="true" rid='12321'>BASE64ENCODED...</key>
    <!-{}- ... -{}->
    <iv>BASE64ENCODED...</iv>
  </header>
</encrypted>
```

This KeyTransportElement can then be sent over any applicable transport mechanism.

4.7 Receiving a message

When an OMEMO element is received, the client MUST check whether there is a <key> element with an rid attribute matching its own device ID. If this is not the case, the element MUST be silently discarded. If such an element exists, the client checks whether the element's contents are a PreKeySignalMessage.
If this is the case, a new session is built from this received element. The client SHOULD 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 delete the private key belonging to the PreKey after use.
If the element's contents are a SignalMessage, and the client has a session with the sender's device, it tries to decrypt the SignalMessage using this session. If the decryption fails or if the element's contents are not a SignalMessage either, the OMEMO element MUST be silently discarded.
If the OMEMO element contains a <payload>, it is an OMEMO message element. The client tries

to decrypt the base64 encoded contents using the key and the authentication tag extracted from the <key> element. If the decryption fails, the client MUST silently discard the OMEMO message. If it succeeds, the decrypted contents are treated as the <body> of the received message.
If the OMEMO element does not contain a <payload>, the client has received a KeyTransportElement. The key extracted from the <key> element can then be used for other purposes (e.g. encrypted file transfer).

5 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 PreKeySignalMessage 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.
As there are no explicit error messages in this protocol, if a client does receive a PreKeySignalMessage using an invalid PreKey, they SHOULD respond with a KeyTransportElement, sent in a <message> using a PreKeySignalMessage. By building a new session with the original sender this way, the invalid session of the original sender will get overwritten with this newly created, valid session.
If a PreKeySignalMessage is received as part of a Message Archive Management (XEP-0313) [7] catch-up 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 MAM catch-up is completed. If this is done, the client MUST then also send a KeyTransportMessage using a PreKeySignalMessage before sending any payloads using this session, to trigger re-keying. (as above) This practice can mitigate the previously mentioned race condition by preventing message loss.
As the asynchronous nature of OMEMO allows decryption at a later time to currently offline devices client SHOULD include a Message Processing Hints (XEP-0334) [8] <store /> hint in their OMEMO messages. Otherwise, server implementations of Message Archive Management (XEP-0313) [9] will generally not retain OMEMO messages, since they do not contain a <body />

[7] XEP-0313: Message Archive Management <https://xmpp.org/extensions/xep-0313.html>.
[8] XEP-0334: Message Processing Hints <https://xmpp.org/extensions/xep-0334.html>.
[9] XEP-0313: Message Archive Management <https://xmpp.org/extensions/xep-0313.html>.

6 Implementation Notes

The SignalProtocol-library uses a trust model that doesn't work very well with OMEMO. For this reason it may be desirable to have the library consider all keys trusted, effectively disabling its trust management. This makes it necessary to implement trust handling oneself.

7 Security Considerations

Clients MUST NOT use a newly built session to transmit data without user intervention. 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.
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.
While it is RECOMMENDED that clients postpone private key deletion until after MAM catch-up and this standards mandates that clients MUST NOT use duplicate-PreKey sessions for sending, clients MAY delete such keys immediately for security reasons. For additional information on potential security impacts of this decision, refer to [10].
In order to be able to handle out-of-order messages, the SignalProtocol stack has to cache the keys belonging to "skipped" messages that have not been seen yet. It is up to the implementor to decide how long and how many of such keys to keep around.

8 IANA Considerations

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

9 XMPP Registrar Considerations

9.1 Protocol Namespaces

This specification defines the following XMPP namespaces:

- eu.siacs.conversations.axolotl

[10] 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.

9.2 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.

10 XML Schema

```
<xml version="1.0" encoding="utf8">
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="eu.siacs.conversations.axolotl"
    xmlns="eu.siacs.conversations.axolotl">

  <xs:element name="encrypted">
    <xs:element name="header">
      <xs:attribute name="sid" type="xs:integer"/>
      <xs:complexType>
        <xs:sequence>
          <xs:element name="key" type="xs:base64Binary" maxOccurs="
              unbounded">
            <xs:attribute name="rid" type="xs:integer" use="required"/
                >
            <xs:attribute name="prekey" type="xs:boolean"/>
          </xs:element>
          <xs:element name="iv" type="xs:base64Binary"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="payload" type="xs:base64Binary" minOccurs="0"/>
  </xs:element>

  <xs:element name="list">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="device" maxOccurs="unbounded">
          <xs:attribute name="id" type="integer" use="required"/>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="bundle">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="signedPreKeyPublic" type="base64Binary">
          <xs:attribute name="id" type="integer"/>
        </xs:element>
```

```
        <xs:element name="signedPreKeySignature" type="base64Binary"/>
        <xs:element name="identityKey" type="base64Binary"/>
        <xs:element name="prekeys">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="preKeyPublic" type="base64Binary"
                  maxOccurs="unbounded">
                <xs:attribute name="id" type="integer" use="required"/
                    >
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

11 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.