Abstract

The history of secure messaging in XMPP is spotty. The long-running de facto scheme, OTR, enjoys fairly wide implementation and use, but OTR suffers from some serious usability and security shortcomings that make it unsuitable as a basis for encryption.

This document describes an architecture that provides end-to-end confidentiality and integrity for XMPP messaging. Solutions for both multi-user and one-to-one messaging are provided.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 1, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

Thomson & Roach Expires April 1, 2016
include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ........................................... 3
2. Goals .................................................. 3
3. Architectural Overview ................................. 3
   3.1. The New Pieces .................................... 4
   3.2. One-to-One Messaging ............................... 4
      3.2.1. Publishing Key Exchange Data ................ 4
      3.2.2. Establishing Pairwise Keying Material ........ 5
      3.2.3. Message Encryption ............................ 5
   3.3. Multi-User Chat .................................. 5
      3.3.1. Inviting Other Clients ....................... 7
4. Identity Assertions .................................... 8
   4.1. Acquiring Identity Assertions .................... 9
   4.2. Validating Identity Assertions ................... 10
5. Message Encryption Details ............................ 11
   5.1. Encapsulation .................................... 11
   5.2. Symmetric Encryption ............................. 13
      5.2.1. Nonces ...................................... 13
      5.2.2. Symmetric Algorithm Agility ................ 13
      5.2.3. Message Drop Attacks ........................ 13
   5.3. Signature ........................................ 14
   5.4. Decryption and Validation ........................ 14
   5.5. Presence Encryption ................................ 15
   5.6. Chat State Notifications .......................... 15
   5.7. Layers of Encryption ............................. 15
6. Key Advertisement ....................................... 15
   6.1. Asymmetric Key Advertisement ..................... 16
   6.2. Symmetric Key Advertisement ..................... 16
   6.3. Key Identifiers .................................. 17
   6.4. Key Lifetime ..................................... 18
7. User and MUC Rosters ................................... 18
   7.1. Roster Entries .................................... 19
      7.1.1. Tracking Affiliations and States ............. 21
      7.1.2. MUC Invitation Tickets ....................... 21
      7.1.3. User Roster Management ....................... 22
      7.1.4. Roster Update Protocol ....................... 22
   7.2. Roster State ...................................... 23
   7.3. Roster Security Limitations ....................... 23
   7.4. Mitigating Attacks ................................ 24
8. Pseudonymity ........................................... 24
9. Acknowledgements ....................................... 25
10. Security Considerations .............................. 25
11. IANA Considerations .................................. 26
1. Introduction

The history of secure messaging in XMPP is spotty. The long-running de facto scheme, OTR, enjoys fairly wide implementation and use, but OTR suffers from some serious usability and security shortcomings that make it unsuitable as a basis for encryption. It also has a number of limitations that make it unsuitable for advanced scenarios, such as multi-user conferences.

This document describes an architecture that provides end-to-end confidentiality and integrity for XMPP messaging. Solutions for both multi-user and one-to-one messaging are provided.

2. Goals

This is a very simple proposition with a somewhat involved solution:

- one-to-one messaging is secured end-to-end
- multi-user chats are secured end-to-end
- messages can be sent to offline users
- the set of entities that can decrypt a message can be audited
- users are able to control whether their communications can be correlated across different venues

3. Architectural Overview

This system aims to achieve the above goals by adding encryption to chat-related XMPP functions.

This only aims to protect chat-related messaging. It provides only limited protection for presence information. The key agreement parts of this protocol are intended to be generically applicable, but the application to file transfer, jingle, and myriad other XMPP features is left for future efforts.
3.1. The New Pieces

Identity assertions (Section 4) allow users to strongly authenticate others. The identity assertion mechanism is also used as the basis of public key distribution, which underpins several of the other building blocks.

Encrypting a message (Section 5) is rather straightforward once the symmetric encryption key is chosen. Establishing the key used in message encryption is more difficult. This design uses a scheme whereby encryption keys are advertised (Section 6) prior to use. How this is done varies only slightly between one-to-one messaging and MUC.

Ensuring proper key distribution of the message-encrypting keys to a potentially large and changing set of users is the most challenging and involved piece of infrastructure to design and build. This architecture uses a secure roster (Section 7) that makes and verifies strong cryptographic assertions about participation.

Finally, pseudonymity functions allow a user to safeguard their privacy. The addition of strong cryptography makes it easier for passive observers to correlate activity, but pseudonymity (Section 8) allows users to minimize what information about their activity is leaked to others.

3.2. One-to-One Messaging

Message encryption for one-to-one messaging uses a unilateral key declaration for key management.

3.2.1. Publishing Key Exchange Data

Clients that wish to participate in encrypted messaging publish keying material to their presence. This includes a signing public key that is used to authenticate messages and a Diffie-Hellman (DH) share used for exchanging the symmetric keys used in encryption. Each client generates new keying material that is bound to the full JID that they use (that is, each client has its own keying material; there is no key associated with a user’s bare JID).

Keys are published under a randomly generated identifier that is consequently used to identify the key when it is used to encipher a message.

New clients can only be added if an existing client attests to the addition. This is intended to stop a subverted server from adding
clients. This uses a form of the same roster log (Section 7) that is used for multi-party chats.

ISSUE: does this represent a UX issue? Five years ago, we might have been concerned about users forgetting a password or losing their last device. Today, this is less of a concern, especially if we subscribe to the principal device theory (NOTE: The principal device here is a smartphone. That device goes with users everywhere and can serve as an anchor for security operations, like adding a new client to the set of authorized clients for a JID.).

3.2.2. Establishing Pairwise Keying Material

Prior to sending a message, a client first retrieves and validates the presence of the intended recipient. A client that supports encryption will include a valid DH share in their presence information.

The sending client then generates new symmetric keys that it will use with this peer. This key is enciphered toward all the clients in the recipient’s presence.

The key is also enciphered toward other authorized clients for the sender’s JID. This allows other clients to decipher the messages that other clients have sent. It also allows those clients to reuse the key.

It is also necessary for clients to re-send this information if a client is added by sender or recipient. Otherwise, clients that reuse symmetric keys will generate messages that new clients are unable to decrypt.

3.2.3. Message Encryption

Once keying material has been selected or new keying material has been advertised, messages are encrypted and decrypted (Section 5) using that symmetric key.

Recipients of the message recover the advertised keying material by retrieving the presence of the sender and decrypting the enciphered key.

3.3. Multi-User Chat
A user founds a MUC in the usual fashion (see [XEP-0045], section 10.1). Two changes are made:

1. The client creates a temporary JID (Section 8) that is unique for the room, and matching keying material that it will use exclusively with that temporary JID.

2. The message that founds the room includes a founding entry for the secure roster log (Section 7) for the room. This is an element that establishes the creating user as an owner in a manner that can be independently verified.
All subsequent changes to the roster of the MUC need to be accompanied by a message that authorizes the change. This message is signed by the user that proposes the change. All users verify the resulting series of changes that accumulate to build up the room roster.

Unauthorized changes to the roster are therefore detectable. Keying material is only shared with users that have been legitimately added to the roster.

### 3.3.1. Inviting Other Clients

In order to invite a user to a chat, two pieces of identifying information for the invited client need to be retrieved: a temporary JID and the keying material for that client.

The inviting client generates a signed invitation and sends this to the bare JID of the offline user. This invitation is a bearer token that can be exercised by any client that has it. The invitation must be encrypted using one-to-one message encryption, or servers can steal and use it. A user with the bearer token includes that in a signed roster log entry when they join the room. The room adds the entry to the roster log if it can be validated.
4. Identity Assertions

The identity of users is one of the most important pieces of confidential information in the context of a chat. Identity information need to be confidentiality protected if they transit more than one server hop.

A new <message> payload is defined to carry identity assertions. That assertion binds a bare JID to the signing public key used by a client to send messages.
The identity assertion contains only a single piece of public information: the domain name of the asserting entity. The remainder is an opaque blob of data that is consumed by the identity provider.

```xml
<message from="..." to="..." id="..." type="chat">
  <x xmlns="...identity#assertion">
    <domain>example.com</domain>
    <assertion>
      Gv3JuuWcMW0NZZib8pk+ZMPS4jnkmt0cFZQT0UT0yAktmseWAk2w
    </assertion>
  </x>
</message>
```

### 4.1. Acquiring Identity Assertions

An identity assertion is acquired from the domain that is responsible for the JID (that is usually the client’s own server) by sending a query to that server. The server uses the client’s authentication credentials, which are usually bound to a connection, to determine if the client owns the identifier.

```xml
<iq from="user@example.com/resource" to="example.com" id="..." type="get">
  <x xmlns="...identity">
    <publicKey>...</publicKey>
  </x>
</iq>
```

The information that is signed is the signing key that the client intends to bind to their identity.

The assertion that the server generates will ultimately be consumed by the server that generated it, so it can be completely opaque. However, it should contain enough information for the server to identify the JID of the client that it relates to, as well as verify its authenticity.

```xml
<iq from="example.com" to="user@example.com/resource" id="..." type="result">
  <x xmlns="...identity#assertion">
    <domain>example.com</domain>
    <assertion>
      Gv3JuuWcMW0NZZib8pk+ZMPS4jnkmt0cFZQT0UT0yAktmseWAk2w
    </assertion>
  </x>
</iq>
```
The assertion might also include limits on validity, such as an expiration time, as dictated by server policy.

4.2. Validating Identity Assertions

ISSUE: This validation mechanism relies on transitive trust in the server of the client making the query. Confidentiality protection seems like the right thing here. Which is a second case for server-to-client confidentiality.

Clients that receive the identity assertion can then query the server that issued it and request the identity that it contains. The server validates the assertion, and either generates an error, or a message containing the identity.

The query includes the assertion:

```xml
<iq from="other@example.net/check" to="example.com" id="..." type="get">
  <x xmlns="...identity#assertion">
    <domain>example.com</domain>
    <assertion>
      Gv3JuuWcMW0NZZib8pk+ZMPS4jnkmt0cFZQTPOTUM0yAktmseWak2w
    </assertion>
  </x>
</iq>
```

A successful response includes the identity of the client, and the public key that was asserted.

```xml
<iq from="example.com" to="other@example.net/check" id="..." type="get">
  <x xmlns="...identity#identity">
    <identity>user@example.com</identity>
    <publicKey>...</publicKey>
  </x>
</iq>
```

A client receiving this response checks that the domain part of the identifier matches the server identity. Once this check is complete, the identity can be associated with the public key. All messages sent with that public key can thereafter be attributed to the identifier. Clients might also provide indicators that the sender of authenticated messages has been verified.
5. Message Encryption Details

A simple combination of symmetric encryption and asymmetric signing are used to protect messages.

This wrapping scheme takes an unencrypted, serialized XMPP stanza. The process adds a signature over this data. Then the signed content is encrypted.

```
content = cleartext || sender.sign(cleartext)
ciphertext = encrypt(keys[keyid].key, nonce, content)
```

The routing and message handling information from the cleartext (element basename plus to, from, and type attributes) is added to a new stanza of the same basename as the original. That is, `<iq>` stanzas produce encrypted `<iq>` stanzas; `<message>` stanzas result in encrypted `<message>` stanzas. Unfortunately, these attributes govern message delivery in ways that could cause compatibility issues if they were encrypted.

**ISSUE:** The potential variations in these values leaks information; a future study might identify mappings that allow for reductions in this leakage. This might include identifying cases where removing the resource identifier from routing attributes is safe; or finding ways to map the range of stanza elements and type attributes to a reduced set. If there was a consistent set of policies with respect to handling the different stanzas and types, this would be easier.

A new `<e>` element is added to the encrypted element. The content of this element is a base64 encoded string that contains the encrypted message.

This element includes attributes for a key identifier (Section 6.3) and sequence number. The key identifier provides the information a recipient needs to decrypt the message. The sequence number increases by one for every message sent, allowing a receiver to detect when messages are dropped or lost.

5.1. Encapsulation

Encrypted messages always use either the `<message>` stanza or the `<iq>` stanza based solely on the nature of the exchange. Messages that require a response from a specific client use the `<iq>` stanza; all other messages use `<message>`.

Presence information can be encrypted, but this is necessarily mixed with unencrypted data. An extension to the `<presence>` element
includes confidential presence information. Note that presence information is effectively broadcast; but any encrypted information will need a limited audience, and all that audience will need to receive the same encryption key.

For example, the following message from [RFC6121] section 5.2.1:

```
<message
  from='juliet@example.com/balcony'
  id='ktx72v49'
  to='romeo@example.net'
  type='chat'
  xml:lang='en'>
  <body>Art thou not Romeo, and a Montague?</body>
</message>
```

Might be encrypted into a message of the form:

```
<message
  from='juliet@example.com/balcony'
  id='ktx72v49'
  to='romeo@example.net'
  type='chat'>
  <e xmlns='...tbd...' key='MSPw9g5t1j9BZGF6' seqno='1'>
    Ume+Bl0ftGYSA2Z2yJMycNvMjmpysdfQY2wcrCiK0AtsrqWZR6KcTTEkewfepW1FGIYpmZFLGSybwRZ+VcOHdi0I9aYVzdSPDmXrM2mrJGhz7sxphlKfPlw6ZrJ7DtGq9IM0epBu6E4hb9DBd4+0RR2Ap7+cAD+ICMJQMySuq/mVE7ybxxWzloU30LbGn/1zU9cmUww3yCI98W2AcHoeTQXJ0b/qjUpqYjJwCshaCH_HH7daksOTS3IoJH0ancJmsBd8RYqMBekrpD9RpuWGGzR-ro0ScRbdsLnzXmY162qSnyw1qCMNuJE5o5_uraBRgEkbCD1Xas
  </e>
</message>
```

This removes the language indicator from the unencrypted stanza.

ISSUE: What do existing clients do when they see an encrypted message? The addition of one client that supports encryption causes encrypted messages to be sent to that user. Other clients that haven’t yet been upgraded will have to deal with the encrypted messages somehow. It might be possible to add a `<body>` element with a generic message, but that would need to be internationalized and repeating that block in every message seems wasteful. Ideally, encrypted messages would just be discarded. To do: check with someone who might know.
5.2. Symmetric Encryption

Authenticated Encryption with Additional Data (AEAD) [RFC5116] is used to provide confidentiality of messages, as well as integrity against unauthorized recipients.

The AEAD key is either advertised (Section 6) or reused from a prior advertisement. The advertisement of the key establishes the scheme that is used.

5.2.1. Nonces

The sequence number on each message determines the nonce that is used with the AEAD. For a given combination of sender and key identifier, sequence numbers cannot repeat without risking compromise of the confidentiality and integrity provided by authenticated encryption. The following nonce derivation method is used (using HKDF as defined in [RFC5869]):

nonce = HKDF(0, sender.fullJID, 'nonce', N_MAX) XOR seqno

This nonce selection ensures a negligible probability of nonce reuse as long as each sender correctly increments the sequence number. Recipients can verify that sequence numbers are not reused.

5.2.2. Symmetric Algorithm Agility

Key identifiers also identify the AEAD algorithm that is used to encipher a message. That information is carried in the key advertisement.

Messages may be enciphered multiple times with different keys. This allows new encryption schemes to be deployed, at the cost of sending some messages multiple times. This is only necessary if some potential recipients only support old AEAD algorithms.

This presents a downgrade attack vector if an attacker can convince a sender that a legitimate client supports a weaker cipher suite. However, key advertisements are authenticated, so the only point for a potential downgrade is a break in the signing key that a recipient uses, which is much more serious a problem than a mere downgrade.

5.2.3. Message Drop Attacks

Selectively dropping messages can be used by an attacker to disrupt communications. Even without visibility into the contents of messages, side channels might be used to learn of the timing of important messages.
For this reason, the sequence number of messages is integral to the encryption of messages. Sequence numbers increase at a constant interval for messages sent by the same client, allowing dropped messages to be detected. Since there is an expectation of reliability and in-order message delivery, clients should highlight where message are missing.

5.3. Signature

A signature on messages is necessary to prevent impersonation of other MUC participants. This means that repudiation of the form that OTR claims to provide is not offered, because that requirement is incompatible. Note however that a temporary MUC using a temporary JID (Section 8) and no identity assertion (Section 4) provides only circumstantial means of attributing activity to a user.

5.4. Decryption and Validation

The reverse of this process is used to decrypt messages. Encrypted information is authenticated and the signature validated. The decrypted and verified stanza is then parsed as though it were in place of the current stanza.

A client only needs to decrypt one <e> element, since each is required to include the same content. All unencrypted content in the stanza is removed and consequently ignored.

It is important that the receiver check that the stanza is whole and valid before allowing it to be processed further. A server that is unable to decrypt a message cannot be relied upon to ensure that messages are valid.

In the cleartext protocol, framing issues do not propagate easily, since they directly affect stanza processing. Encrypted stanzas allow a malicious peer to generate invalid - especially unterminated - XML. Extraneous bogus frames resulting from unchecked XML might be exploited to impersonate a server toward a receiving client. Matching the enciphered *from* attribute against the included signature is also necessary to prevent other forms of impersonation.

Additional checks might be necessary for specific stanzas, types, or content. In general, any checks that might have been possible on a server need to be carried out by clients that received encrypted stanzas.
5.5. Presence Encryption

Some presence information might be confidential. For instance, many users include a status message that is shared with their friends. Encrypting status is highly desirable.

Direct children of the presence stanza may be encrypted in an <e> element. These necessarily use a different key than those used for other types of messaging to avoid problems with controlling key disclosure.

ISSUE: we need to expand on the definition of presence. This could require new messaging arrangements to support retrieval of encrypted presence information. It definitely needs more expansion on key management. At its core, this might reuse the same systems used for one-to-one messaging, but the broadcast nature of presence means that there is a larger audience for any given message, which increases the potential for keys to be unavailable to valid recipients.

5.6. Chat State Notifications

Clients are required to encrypt chat state notifications [XEP-0085]. However, these messages are useless to an offline client. A server that can see these messages is required to drop them (see [XEP-0160], section 3), but encrypted messages can’t be distinguished from other more important messages.

Thus, clients are required to suppress chat state notifications when a peer is offline. This allows a server to fake presence for a user in order to filter out chat state notifications, or suppress the feature, but the value of this information isn’t great and this causes an attacker to actively modify or suppress messages to mount the attack.

5.7. Layers of Encryption

A group chat might use one-to-one message encryption to send messages to a user. There’s a question about what advantage that provides. Removing the nick from messages might be of some advantage, but that advantage is better managed with Section 8.

6. Key Advertisement

Users publish the encryption keys that they use for one-to-one messaging to their presence. In an MUC, they instead use the presence that they advertise to the room.
6.1. Asymmetric Key Advertisement

Two forms of asymmetric keys are used by clients: a signing key that is used to authenticate all forms of messages sent by that client, and a Diffie-Hellman (likely elliptic curve) share that is used in symmetric key advertisements.

Asymmetric signing keys are added to and form the basis of the roster log (Section 7) maintained for each user or MUC. Diffie-Hellman shares are added to presence information.

Diffie-Hellman shares are added to a presence document as JWS-signed [RFC7515] JWK Set [RFC7517]. Each JWK in the set includes a key identifier that other clients use to identify the key.

```xml
<presence from="jid@example.com/resource">
  <x xmlns="....:shares">
    {
      "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9MA4MTkzODAsDQogImh0dHA6Ly9MA4MT10",
      "signatures": [
        {"protected": "eyJhbGciOiJ1NiJ9",
         "header": { "jwk": { ... } },
         "signature": "cC4hiUPoj9Eetdgtv3hF80EGrhubE_dzERat0XF9g2VtQgr9Pjbo3XOicJy5RZmh7AuH1m4Bh-0Qc_1F5YKt_08W2Fp5juGbd9uJrbF9CUAr71dnZcAcoJbKBYNXI4BAynRFd1uB--f_nZLGmby6yWz0Z75vRKh6xBAirLIARPNvksJtQBMHl1L07Qe7K0Gar2RmB_eSN9383LoOLn6_dO--xil2jzDwsc-eO6k4W
EsqtFZEC56BF17noOPqvhJ1phCnvWkH6lEyY1z9Q0YE4upUI18np6LbgGY9F
s98rqVt5AXLIhWkWYw1VmtVrB0igcN_IoypGlUPQGe77Rw"
        }
      ]
    }
  </x>
</presence>
```

Each key that is encrypted toward this client uses the "kid" parameter on the JWK to identify the key that they are encrypting towards.

Diffie-Hellman advertisements are signed to ensure that they are only provided by authorized clients. This allows these advertisements to be generated prior to a client becoming authorized.

6.2. Symmetric Key Advertisement

A key advertisement contains the following information:

- A key identifier
An encryption scheme identifier

The full JID of the advertising client (a temporary JID for MUC)

The full JID of the key recipient (a temporary JID for MUC)

A key identifier for the DH share for the advertising client

A key identifier for the DH share for the key recipient

A symmetric key, encrypted using a key derived from the DH exchange

An expiration date and time, after which the key must not be used for enciphering new messages, though it may be used to decipher old messages

A key identifier for the signing key

A signature over this entire structure, generated using the private signing key

Open Issue: Rather than use key identifiers for DH shares it might be easier to exchange the share itself, since that is unambiguous and likely not significantly different in size. That makes the advertisement self-contained. Note that this would not absolve the key recipient of the need to check that the DH share (or signature key) is from a valid and authorized entity. The use of a key identifier makes that check implicit and avoids some types of mistake.

The signature is required to ensure that a key is not replayed and consequently reused. A message with a signing key that is either not known or from an entity that is not permitted to participate in a conversation must be discarded.

A complete key advertisement includes the same information repeated for each recipient. Common information, and the signature, don’t need to be repeated. A single signature has implications for key lifetime.

6.3. Key Identifiers

Key identifiers are used to select the key that is used for encryption and decryption. Each key advertisement has an associated key identifier.
Care needs to be taken to ensure that key identifiers are unique within the context that they are created. Since keys are proposed and used by multiple actors without synchronization, identifying keys with a large identifier (such as a GUID) is advised.

6.4. Key Lifetime

Keys are advertised with an expiration time that limits the time when they can be used for encryption. However, offline clients need to be able to read messages that are generated while they are offline. Clients that are offline for extended periods need to be able to recover the keys that were used to encrypt those messages.

Keys advertised to user presence are therefore persisted until their intended recipients have retrieved and acknowledged keys. Since each key is encrypted toward a specific client, once that client retrieves the key it can be removed, though explicit acknowledgment might be desirable. Note that the set of recipients includes all client instances of the intended recipient, plus all client instances of the sender.

Once the complete set of potential recipients have acknowledged a key, then it can be removed. This might use implicit acknowledgment for client instances of the sender, since the server can track message delivery to those clients. Explicit acknowledgment is necessary for remote clients of the recipient.

Keys advertised within an MUC enter the chat transcript. New messages to the chat are expected to use the latest key, so old keys only need to be maintained to account for race conditions where messages might be sent without knowledge of the most recent key. Keys that are superseded by a newer key can therefore be disposed of after a short duration.

Here, the ordering of messages to the MUC is used to determine which key is used. That ordering allows clients to remove and discard older keys.

7. User and MUC Rosters

All efforts to encipher messages are largely pointless if the architecture permits servers to add themselves to the set of clients and thereby acquire keying material. The roster of clients that are authorized to represent a user, or which are part of an MUC, is a resource that needs strong integrity protection to prevent a malicious server from becoming part of conversations.
The canonical form of each roster takes the form of a log. A roster log is a verifiable chain of changes to the roster. The log can be validated by any entity and the set of participants validated.

Each entry in the log identifies the entry that immediately precedes it by including a cryptographic hash of that entry. This ensures that a valid log cannot include divergent or conflicting changes (this does not prevent certain forms of manipulation (Section 7.3) by servers). Each entry is signed by the entity that generated the entry, allowing changes to be attributed and validated.

A successfully validated roster log can be used by a client to determine the set of clients that a key advertisement (Section 6) needs to be enciphered toward. In ensuring that a validated roster log is used prior to advertising new keys, clients can ensure that only authorized clients receive those keys.

7.1. Roster Entries

There are several types of entry that can be recorded into the roster log. The following table summarizes the different types of log entry.
<table>
<thead>
<tr>
<th>Entry Type</th>
<th>Parameters</th>
<th>Who can add</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Room Type</td>
<td>Room Owner, Room Type</td>
<td>Room Owner</td>
<td>This entry must be the first entry in a MUC roster log. This also establishes the signer as a room owner. (MUC only)</td>
</tr>
<tr>
<td>Set Room Permissions</td>
<td>Permissions</td>
<td>Room Owner</td>
<td>The set of permissions are taken from [XEP-0045], limited to those that can change. (MUC only)</td>
</tr>
<tr>
<td>Set Affiliation</td>
<td>Subject, Affiliation</td>
<td>Authorized Client</td>
<td>The maximum role that the subject can assume might be included, or it might be determined based on [XEP-0045] affiliation. (MUC only)</td>
</tr>
<tr>
<td>Set Client State</td>
<td>Subject, State</td>
<td>Authorized Client</td>
<td>This determines whether the identified client is authorized or not. (One-to-one only)</td>
</tr>
<tr>
<td>Redeem Ticket</td>
<td>Subject, Invitation Ticket</td>
<td>Subject</td>
<td>This in effect allows for a Set Affiliation entry to be generated in two parts.</td>
</tr>
<tr>
<td>Rekey</td>
<td>Old Key, New Key</td>
<td>Subject</td>
<td>Used by a client to replace the keying material it uses without changing the affiliation of the client. This entry is signed with the old key. This invalidates the old key for future use.</td>
</tr>
</tbody>
</table>

Subjects are identified in the roster log by their signing public key.
Each log entry includes a hash of the message that precedes it. This ensures that a log entry cannot be replayed on top of a different roster state.

The owner (see [XEP-0045], section 5.2) affiliation is required to perform important tasks, like setting the room type or altering configuration options. For setting the affiliation or state of other clients, a combination of factors determine whether an operation is valid: the affiliation of the user performing the change, the room type, and the room permissions.

7.1.1. Tracking Affiliations and States

The roster log becomes the source of truth for affiliations (or for client state). This has a range of consequences, some of which result in divergence from unprotected chat or MUC.

The roster log is public information. This means that affiliations and states can be seen by any client. This information is advertised to room participants in the presence advertisements, but rooms can be configured to suppress presence. Access to affiliation information for offline users (member, admin, and owner lists) is controlled.

The design to this point assumes that there is some functional distinction between an affiliation of Member and an affiliation of None. This is not the case in an Open or Unmoderated room. For those room types, the roster log does not need to track affiliation transitions between Member and None, though it may if the room type could change.

**ISSUE:** Is there any sense in encrypting communications when a room is open or unmoderated? Since anyone can join, the main value is in having some knowledge about the set of clients that might have received a message. Given the high level of pseudonymity used, is even that much achievable?

The Outcast affiliation cannot be tracked, which makes it impossible to ban a user from an MUC. That is not a consequence of the roster log design, but a result of requiring the use of pseudonyms (Section 8) in MUC.

7.1.2. MUC Invitation Tickets

An offline client or user is invited to an MUC by sending them an invitation ticket.
An invitation ticket contains all the information that a Set Affiliation roster log entry might have, without a subject. That subject is provided when the ticket is redeemed.

- Affiliation (and role)
- GUID (or equivalent randomness)
- A signature from a client that is authorized to make the change

Invitation tickets are bearer tokens. That means that distribution of these messages needs to use confidentiality protection, such as that provided by secure one-to-one messaging.

A client can revoke any unused tickets that they have sent by rekeying. Consequently, new invitations have to be issued if a client updates their keys.

7.1.3. User Roster Management

It might seem obvious that an equivalent ticket mechanism could be used for managing user rosters. A user that has a new device, could send that device an invitation to join their list of clients.

However, offline invitations for alternative clients don’t have a confidentiality protection mechanism available: MUC invitations can be one-to-one encrypted because the user roster exists and provides for that confidentiality protection. The same capability is not available for the client roster, which introduces a bootstrapping problem.

The safest option here is to leave this unspecified and require that clients add other clients to the user roster directly. That means that new clients need to come online and advertise keys before they can be invited to represent a client.

7.1.4. Roster Update Protocol

A simple IQ protocol is defined for updating the roster. A client sends a new roster element (a JWS signed JSON structure) to the server that maintains the roster as follows:
<iq from="user@example.com/resource" to="muc@example.com" type="set">
   <x xmlns="...:roster">
      ... a JWS-signed blob ...
   </x>
</iq>

The server can validate this and add it or not based on that validation. Note that this exposes clients to various forms of denial of service mounted by the browser, primarily a refusal to take valid updates.

7.2. Roster State

Clients process a roster log to produce their own view of the state of the roster. This ultimately results in a set of clients that are authorized to receive key advertisements.

Each client maintains their own view of the state of the roster for other clients and each MUC that they participate in. This state can be recovered at any time by re-processing the roster log. Clients use this state to select the clients that keying material can be shared with. Clients also use roster state in determining whether a new roster log entry is valid.

A roster log can enter a broken state if an invalid entry is added. Servers are expected to validate new entries and ensure that this doesn’t happen, but it is possible that errors or malice could cause invalid entries to be recorded and distributed. Clients are required to freeze the state of a roster at the point where the last valid entry is found.

7.3. Roster Security Limitations

We have to assume that an attacker (in particular, the server that maintains and distributes a roster log) can affect how a roster log makes progress.

This can be used to an attacker’s advantage. An attacker can withhold new changes from clients, or from a subset of clients. By preventing some subset of clients from learning about changes, an attacker can freeze the state of a roster from the perspective of those clients.

This could potentially be used to stimulate the creation of multiple different changes from the same starting state. The attacker might then choose to allow changes only that are favorable to it.
In general, this means that the progress of a roster state has to be viewed as a directed graph, not a linear sequence. The nodes of the graph correspond to states of the roster. The outbound edges from any node are the valid set of changes that might be made from that state by any agent that is currently present.

Given sufficient time, a server can direct progress along any edge that is presented to it. Also, the server can freeze the node that an individual client sees by refusing to forward entries to that client.

More opportunities are available to the server if clients rely on the server to maintain the entirety of the log state. A client that maintains no state about a roster opens itself to the possibility that the server could set the state of the roster to any node in the directed graph, including old states.

7.4. Mitigating Attacks

The obvious protection clients can use to limit the potential for unconstrained state manipulation is remembering the state of a log. This can be limited to the last entry (or the hash of that entry), even if clients need to discard other state. This prevents roster progress from being rewound, but it cannot do anything about a server withholding entries that a client hasn’t seen.

The potential for attacks based on withholding log entries is a potentially serious concern. The design of XMPP naturally provides a single central controller: the server. That central controller can provide excellent consistency, if we assume that the server chooses to present the same view of the roster log to all clients. However, if the server is malicious, then it represents a single point of failure.

ISSUE: It might be that this is an acceptable condition, given the limited opportunity that the server has to affect change. An alternative design would decouple the roster management function from the message delivery function, which would allow this to be independent of the XMPP server. That opens other options, like distributed or redundant roster stores, though a decoupled design adds new error conditions for every problem it aims to address.

8. Pseudonymity

Providing end-to-end confidentiality and integrity greatly improves the privacy profile of XMPP. However, exposure of a user’s JID to a group chat server allows for a greater degree of traffic analysis.
This proposes the use of a pseudonym to minimize the information that is made available to a group chat server.

A new service is added whereby a client can request the creation of an unlinked pseudonym. That pseudonym is a temporary JID. A temporary JID is a bare JID that is aliased to another bare JID. Resource identifiers can then be selected by the client so that messages routed to pseudonym can’t be linked to their primary identifier.

A pseudonym allows a client to join a group chat without exposing their identity to the group chat.

A pseudonym request is simple:

```xml
<iq from="user@example.com/resource" to="example.com" id="..." type="get">
   <x xmlns="...pseudonym"/>
</iq>
```

The response is equally simple:

```xml
<iq from="user@example.com/resource" to="example.com" id="..." type="result">
   <x xmlns="...pseudonym">7ee6df7a831198624131@example.com</x>
</iq>
```

Clients are able to include the new pseudonym in any interaction that they initiate and other servers and users will be unable to distinguish that user from any other user at the same server.

Messages for a pseudonym will be routed to the bare JID of the user, subject to the normal rules for routing of messages to a bare JID, even if the message contains a resource identifier.

9. Acknowledgements

TBD

10. Security Considerations

The entire contents of this document deal with matters of security.
11. IANA Considerations

This document makes no requests of IANA

12. References

12.1. Normative References


12.2. Informative References


Authors’ Addresses

Martin Thomson
Mozilla
\
Mountain View, CA
US

Phone: +1 650 903 0800
Email: martin.thomson@gmail.com

Adam Roach
Mozilla
\
Dallas
US

Phone: +1 650 903 0800 x863
Email: adam@nostrum.com