XEP-0031: A Framework For Securing Jabber Conversations

Paul Lloyd
mailto:paul_lloyd@hp.com
xmpp:paul_lloyd@jabber.hp.com(private)

2002-07-09
Version 0.2

Status	Type	Short Name
Deferred	Standards Track	

Although the value and utility of contemporary instant messaging systems, like Jabber, are now indisputable, current security features to protect message data are generally inadequate for many deployments; this is particularly true in security conscious environments like large, commercial enterprises and government agencies. These current features suffer from issues of scalability, usability, and supported features. Furthermore, there is a lack of standardization. We present a protocol to allow communities of Jabber users to apply cryptographic protection to selected conversation data.

Legal

Contents

1 Introduction

Instant messaging has clearly crossed the chasm from experimental to mainstream in a short amount of time. It is particularly interesting to note the extent to which the employees and affiliates of large enterprises have adopted instant messaging as part of their daily professional lives. IM is no longer simply used on Friday evening to select which movie to watch; it's now used on Monday morning to select which company to acquire.
While the benefits of IM are clear and compelling, the risks associated with sharing sensitive information in an IM environment are often overlooked. We need a mechanism that permits communities of users to protect their IM conversations. This document presents an extension protocol that can be incorporated into the existing Jabber protocol to provide such a mechanism. We hope that this protocol spurs both interest and further investigation into mechanisms to protect Jabber conversations. We also hope that the Jabber community can accelerate the adoption of standardized security mechanisms.
In addition to its ability to protect traditional messaging data, the proposed protocol may also serve as a foundation for securing other data transported via other Jabber extensions.
We use the following terms throughout this document to describe the most relevant aspects of the IM environment that we wish to address:

- user. A user is simply any Jabber user. Users are uniquely identified by a JID; they connect to Jabber hosts using a Jabber node.
  Users produce and consume information, and we wish to provide them with mechanisms that can be used to protect this information.

- community. A community is a collection of users who wish to communicate via Jabber. No restrictions or assumptions are made about the size of communities or the geographical, organizational, or national attributes of the members. Communities are assumed to be dynamic and ad-hoc. Users typically join communities by the simple act of invitation. All members of a community are assumed to be peers.
  The members of communities share information among themselves, and we wish to provide them with mechanisms that can permit information to only be shared by community members.

- conversation. A conversation is the set of messages that flows among the members of a community via some network. Conversations consist of both the actual conversation data produced and consumed by the various users as well as the Jabber protocol elements that transport it. Members participate in a conversation when they are the source or destination of this traffic.
  In hostile network environments, like the Internet, conversation data is vulnerable to a variety of well-known attacks.

Other Jabber and IM terms are used in a traditional, intuitive fashion.

2 Requirements And Considerations

The proposed protocol is designed to address the specific requirements and considerations presented in this section.

2.1 Security Requirements

2.1.1 Data Protection Requirements

A secure IM system must permit conversation participants to preserve the following properties of their conversation data:

- confidentiality. Conversation data must only be disclosed to authorized recipients
- integrity. Conversation data must not be altered
- data origin authentication. Recipients must be able to determine the identity of the sender and trust that the message did, in fact, come from the sender. It is important to note that this requirement does not include the requirement of a durable digital signature on conversation data.
- replay protection. Recipients must be able to detect and ignore duplicate conversation data.

These are established, traditional goals of information security applied to the conversation data. In the IM environment, these goals protect against these attacks:

- eavesdropping, snooping, etc.
- masquerading as a conversation participant
- forging messages

Preserving the availability of conversation data is not addressed by this protocol.
Preserving the anonymity of conversation participants is an interesting topic which we defer for future exploration.
Finally, note that this protocol does not concern any authentication between a Jabber node and a Jabber host.

2.1.2 Data Classification Requirements

A secure IM system must support a data classification feature through the use of security labeling. Conversation participants must be able to associate a security label with each piece of conversation data. This label may be used to specify a data classification level for the conversation data.

2.1.3 The End To End Requirement

It is easy to imagine Jabber systems in which the servers play active, fundamental roles in the protection of conversation data. Such systems could offer many advantages, like:

- allowing the servers to function as credential issuing authorities,
- allowing the servers to function as policy enforcement points.

Unfortunately, such systems have significant disadvantages when one considers the nature of instant messaging:

- Many servers may be untrusted, public servers.
- In many conversation communities, decisions of trust and membership can only be adequately defined by the members themselves.
- In many conversation communities, membership in the community changes in real time based upon the dynamics of the conversation.
- In many conversation communities, the data classifaction of the conversation changes in real time based upon the dynamics of the conversation.

Furthermore, the widespread use of gateways to external IM systems is a further complication. Based on this analysis, we propose that security be entirely controlled in an end to end fashion by the conversation participants themselves via their user agent software.

2.1.4 Trust Issues

We believe that, ultimately, trust decisions are in the hands of the conversation participants. A security protocol and appropriate conforming user agents must provide a mechanism for them to make informed decisions.

2.1.5 Cryptosystem Design Considerations

One of the accepted axioms of security is that people must avoid the temptation to start from scratch and produce new, untested algorithms and protocols. History has demonstrated that such approaches are likely to contain flaws and that considerable time and effort are required to identify and address all of these flaws. Any new security protocol should be based on existing, established algorithms and protocols.

2.2 Environmental Considerations

Any new IM security protocol must integrate smoothly into the existing IM environment, and it must also recognize the nature of the transactions performed by conversation participants. These considerations are especially important:

- dynamic communities. The members of a community are defined in near real time by the existing members.

- dynamic conversations. Conversations may involve any possible subset of the entire set of community members.

Addressing these considerations becomes especially crucial when selecting a conference keying mechanism.

2.3 Usability Requirements

Given the requirement to place the responsibility for the protection of conversation data in the hands of the participants, it is imperative to address some fundamental usability issues:

- First, overall ease of use is a requirement. For protocol purposes, one implication is that some form of authentication via passphrases is necessary. While we recognize that this can have appalling consequences, especially when we realize that a passphrase may be shared by all of the community members, we also recognize the utility.

- PKIs are well established in many large organizations, and some communities will prefer to rely on credentials issued from these authorities. To ensure ease of use, we must strive to allow the use of existing PKI credentials and trust models rather than impose closed, Jabber-specific credentials.

- Finally, performance must not be negatively impacted; this is particularly true if we accept that most communities are composed of human users conversing in real time. For protocol purposes, one obvious implication is the desire to minimize computationally expensive public key operations.

We note that, in practice, the design and construction of user agents will also have a major impact on ease of use.

2.4 Development And Deployment Requirements

To successfully integrate into the existing Jabber environment, an extension protocol for security must satisfy the following:

- It must be an optional extension of the existing Jabber protocol.

- It must be transparent to existing Jabber servers.

- It must function gracefully in cases where some community members are not running a user agent that supports the protocol.

- It must make good use of XML.

- It must avoid encumbered algorithms.

- It must be straightforward to implement using widely available cryptographic toolkits.

- It must not require a PKI.

Failure to accommodate these will impede or prohibit adoption of any security protocol.

3 Protocol Specification

3.1 Protocol Overview

Ultimately, conversation data is protected by the application of keyed cryptographic operations. One operation is used to provide confidentiality, and a separate operation is used to provide integrity and data origin authentication. The keys used to parameterize these operations are called conversation keys. Each conversation should have its own unique set of conversation keys shared among the conversation participants.
Conversation keys are transported among the conversation participants within a negotiated security session. A security session allows pairs of conversation participants to securely share conversation keys throught all participants in the conversation as required.

3.2 Definitions And Notation

The following terms are used throughout this specification:

- initiator. The initiator is the user who requested a security session negotiation. Initiator's are identified by their JID.

- responder. The responder is the user who responded to a security session negotiation request. Responder's are identified by their JID.

- hmac. This indicates the HMAC algorithm. The notation hmac (key, value) indicates the HMAC computation of value using key.

- concatentation operator. The '|' character is used in character or octet string expressions to indicate concatenation.

- security session ID. A character string that uniquely identifies a security session between two users. Security session IDs MUST only consist of Letters, Digits, and these characters: '.', '+', '-', '_', '@'. Security session IDs are case sensitive.

- SS. This term indicates the security session secret that is agreed to during a security session negotiation.
- SKc. This term indicates the keying material used within a security session to protect confidentiality. The SKc is derived from the security session secret, SS.
- SKi. This term indicates the keying material used within a security session to protect integrity and to provide authnetication. The SKi is derived from the security session secret, SS.
- conversation key ID. A character string that uniquely identifies a conversation key shared by a community of users. Conversation key IDs MUST only consist of Letters, Digits, and these characters: '.', '+', '-', '_', '@'. Conversation key IDs are case sensitive. Conversation key IDs SHOULD be generated from at least 128 random bits.
- passphrase ID. A character string that uniquely identifies a passphrase shared by a community of users. Passphrase IDs MUST only consist of Letters, Digits, and these characters: '.', '+', '-', '_', '@'. Passphrase IDs are case sensitive.

3.3 XML Processing

Since cryptographic operations are applied to data that is transported within an XML stream, the protocol defines a set of rules to ensure a consistent interpretation by all conversation participants.

3.3.1 Transporting Binary Content

Binary data, such as the result of an HMAC, is always transported in an encoded form; the two supported encoding schemes are base64 and hex.
Senders MAY include arbitrary white space within the character stream. Senders SHOULD NOT include any other characters outside of the encoding set.
Receivers MUST ignore all characters not in the encoding set.

3.3.2 Transporting Encrypted Content

Encrypted data, including wrapped cryptographic keys, are always wrapped per XML Encryption.

3.3.3 HMAC Computation

HMACs are computed over a specific collection of attribute values and character data; when computing an HMAC the following rules apply:

- All characters MUST be encoded in UTF-8.
- The octets in each character MUST be processed in network byte order.
- For a given element, the attribute values that are HMACed MUST be processed in the specified order regardless of the order in which they appear in the element tag.
- For each attribute value, the computation MUST only include characters from the anticipated set defined in this specification; in particular, white space MUST always be ignored.
- For character data that is represented in an encoded form, such as base64 or hex, the computation MUST only include valid characters from the encoding set.

3.3.4 Performing Cryptographic Operations

The following algorithm is used to encrypt a character string, such as an XML element:

- The character string MUST be encoded in UTF-8.
- The octets in each character MUST be processed in network byte order.
- Appropriate cryptographic algorithm parameters, such as an IV for a block cipher, are generated.

3.4 XML Namespaces

In order to integrate smoothly with the existing Jabber protocol, this protocol utilizes a new XML namespace, jabber:security.

3.5 Security Sessions

3.5.1 Overview

A security session is a pair-wise relationship between two users in which the users have achieved the following:

- They have mutually authenticated each other using credentials acceptable to both.
- They have agreed on a set of key material known only to both.

Security sessions are identified by a 3-tuple consisting of the following items:

- initiator. This is the JID of the user who initiated the session.
- responder. This is the JID of the user who responded to the initiator's request.
- sessionId. A label generated by the initiator.

Security sessions are used to transport conversation keys between the conversation participants.
Scalabilty is an immediate, obvious concern with such an approach. We expect this approach to be viable in practice because:

- The number of participants in typical, interactive conversations is generally on the order of 10^1.
- New participants are usually invited to dynamically join a conversation by being invited by an existing participant; this existing participant is the only one who needs to establish a security session with the new participant, because this single security session can be used to transport all of the required conversation keys.
- User agents can permit the lifetime of security sessions to last long enough to allow transport of conversation keys for a variety of converstions.
- Conversation keys can be established with a suitable lifetime.

Other approaches, including the incorporation of more sophisticated conference keying algorithms, are a topic for future exploration.

3.5.2 Security Session Negotiation

Security sessions are negotiated using an authenticated Diffie-Hellman key agreement exchange. The two goals of the exchange are to perform the mutual authentication and to agree to a secret that is know only to each.
The exchange also allows the parties to negotiate the various algorithms and authentication mechanisms that will be used.
Once the pair agree on a shared secret, they each derive key material from the secret; this key material is used to securely transport the conversation keys, which are used to actually protect conversation data.
The protocol data units (PDUs) that comprise the exchange are transported within existing Jabber protocol elements.

3.5.3 DTDs

```
<!ELEMENT session1
          (nonce, keyAgreement, algorithms, authnMethods) >
<!ATTLIST session1
          version CDATA #REQUIRED
          initiator CDATA #REQUIRED
          responder CDATA #REQUIRED
          sessionId CDATA #REQUIRED
          hmac (hmac-sha1) #REQUIRED >

<!ELEMENT nonce
          (#PCDATA)* >
<!ATTLIST nonce
          encoding (base64 | hex) #REQUIRED >

<!ELEMENT keyAgreement
          (dh) >

<!ELEMENT dh
          (publicKey) >
<!ATTLIST dh
          group (modp1024 | modp2048 | modp4096 | modp8192) #REQUIRED
             >

<!ELEMENT publicKey
          (#PCDATA)* >
<!ATTLIST publicKey
          encoding (base64 | hex) #REQUIRED >

<!ELEMENT algorithms
          (algorithm)+ >

<!ELEMENT algorithm
```

```
          (confAlg, hmacAlg) >

<!ELEMENT confAlg EMPTY >
<!ATTLIST confAlg
          cipher (3des-cbc | aes-128-cbc | aes-256-cbc) #REQUIRED >

<!ELEMENT hmacAlg EMPTY >
<!ATTLIST hmacAlg
          alg (hmac-sha1 | hmac-md5) #REQUIRED>

<!ELEMENT authnMethods
          (authnMethod)+ >

<!ELEMENT authnMethod
          (digSig | passphrase) >

<!ELEMENT digSig
          (certificate+, caCertificate*) >
<!ATTLIST digSig
          alg (rsa) #REQUIRED>

<!ELEMENT certificate
          (#PCDATA)* >
<!ATTLIST certificate
          type (x509 | pkcs7) #REQUIRED
          encoding (base64 | hex) #REQUIRED >

<!ELEMENT caCertificate
          (#PCDATA)* >
<!ATTLIST caCertificate
          type (x509 | pkcs7) #REQUIRED
          encoding (base64 | hex) #REQUIRED >

<!ELEMENT passphrase EMPTY >
<!ATTLIST passphrase
          passphraseId CDATA #REQUIRED >


<!ELEMENT session2
          (nonce, keyAgreement, algorithm, authnMethod, authenticator)
              >
<!ATTLIST session2
          version CDATA #REQUIRED
          initiator CDATA #REQUIRED
          responder CDATA #REQUIRED
          sessionId CDATA #REQUIRED
          hmac (hmac-sha1) #REQUIRED >

<!ELEMENT authenticator
```

```
          (#PCDATA)* >
<!ATTLIST authenticator
          encoding (base64 | hex) #REQUIRED>


<!ELEMENT session3
          (authenticator, keyTransport*) >
<!ATTLIST session3
          version CDATA #REQUIRED
          initiator CDATA #REQUIRED
          responder CDATA #REQUIRED
          sessionId CDATA #REQUIRED
          hmac (hmac-sha1) #REQUIRED >
```

3.5.4 Generating And Sending the session1 PDU

The initiator's user agent employs the following algorithm to generate the session1 PDU:

- Appropriate values for the version, initiator, responder, sessionId, and hmac attributes are assembled. The version of this specification is '1.0'. The values of initiator and responder MUST be the JIDs of the two participants, respectively.

- The nonce is prepared by first generating a string of 20 random octets (160 random bits). The octets are then encoded into a string of 40 hex characters representing the random string.

- A Diffie-Hellman group is selected. The appropriate values for g and p will be used to generate the initiator's public key.

- An ephemeral private key, x, is generated using g and p for the selected group. This key MUST be generated using an appropriate random number source. The corresponding public key, g^x, is generated and encoded.

- The desired set of confidentiality and HMAC cryptographic algorithms is selected. The manner in which these algorithms are selected and all related policy issues are outside the scope of this specification.

- The desired set of authentication algorithms is selected. The manner in which these algorithms are selected and all related policy issues are outside the scope of this specification. When the digital signature form of authentication is selected, the relevant end-entity certificate and, optionally, a chain of CA certificates representing a validation path, is assembled and encoded. A set of trusted CA certificates MAY optionally be included via caCertificate elements; if so, the set MUST include the issuer of the initiator's end-entity certificate.

These values are then used to prepare the XML session1 element; this element is transmitted via the existing Jabber iq mechanism:

```
<iq from="initiator's␣JID" to="responder's␣JID" type="get" id="
   whatever">
   <query xmlns="jabber:security:session">
      <session1>...</session1>
   </query>
</iq>
```

3.5.5 Receiving And Processing the session1 PDU

The responder's user agent employs the following algorithm to process each session1 PDU:

- The version and hmac attributes are checked against the values supported by the user agent. An unsupported version results in an error code of 10000, and an unsupported hmac results in an error code of 10001. The responder attribute MUST match the JID of the receiver; a mismatch results in an error code of 10009

- The nonce is decoded, and its length is checked. The nonce may also be checked to detect replays. An invalid nonce results in an error code of 10002.

- The Diffie-Hellman group is checked against the values supported by the user agent. An unsupported group results in an error code of 10003

- The desired confidentiality and HMAC cryptographic algorithms are selected from the proposed set. The manner in which these algorithms are selected and all related policy issues are outside the scope of this specification. If none of the proposed algorithms are supported, an error code of 10004 occurs.

- The desired authentication algorithm is selected from the proposed set. The manner in which this algorithm is selected and all related policy issues are outside the scope of this specification. In the digital signature case, the responder's end-entity certificate MUST be issued by one of the trusted CAs listed in the session1 PDU or by the same issuer as the initiator's end-entity certificate. If none of the proposed algorithms are supported, an error code of 10005 results. If the responder does not have acceptable credentials, an error code of 10006 occurs.

If any errors occur during processing, the session negotiation fails, and the error is communicated via the existing Jabber iq mechanism:

```
<iq from="responder's␣JID" to="initiator's␣JID" type="error" id="
    whatever">
   <error code="???">...</error>
</iq>
```

If no errors occur, then the responder's user agent proceeds with the session2 PDU.

3.5.6 Generating And Sending the session2 PDU

The responder's user agent employs the following algorithm to generate the session2 PDU:

- Appropriate values for the version, initiator, responder, sessionId, and hmac attributes are assembled. The version of this specification is '1.0'. The values of initiator and responder MUST be the JIDs of the two participants, respectively. The sessionId and hmac values MUST match the sessionId and hmac values contained in the session1 PDU.

- The nonce is prepared by first generating a string of 20 random octets (160 random bits). The octets are then encoded into a string of 40 hex characters representing the random string.

- An ephemeral private key, y, is generated using g and p for the group indicated by the session1 PDU. This key MUST be generated using an appropriate random number source. The corresponding public key, g^y, is generated and encoded.

- The desired pair of confidentiality and HMAC cryptographic algorithms is selected. The manner in which this pair is selected and all related policy issues are outside the scope of this specification.

- The desired authentication algorithm is selected. The manner in which this algorithm is selected and all related policy issues are outside the scope of this specification. When the digital signature form of authentication is selected, the relevant end-entity certificate and, optionally, a chain of CA certificates representing a validation path, is assembled and encoded.

- Based on the selected authentication algorithm, the responder's authenticator is constructed. A digital signature algorithm requires calculating:

  - HK = hmac (initiator's nonce | responder's nonce, g^xy)

  - HASH_R = hmac (HK, version | sessionId | g^y | g^x | responder's JID)

  HASH_R is signed using the responder's private key and encoded in PKCS#1 format. The PKCS#1 octets are then further encoded in base64 or hex.
  The passphrase algorithm requires calculating:

  - HK = hmac (hash (passphrase), initiator's nonce | responder's nonce)

  - HASH_R = hmac (HK, version | sessionId | g^y | g^x | responder's JID)

  The octets of HASH_R are simply encoded in base64 or hex.
  The manner in which the responder's user agent gains access to the responder's credentials is outside the scope of this specification.

These values are then used to prepare the XML session2 element; this element is transmitted via the existing Jabber iq mechanism:

```
<iq from="responder's␣JID" to="initiator's␣JID" type="result" id="
    whatever">
   <query xmlns="jabber:security:session">
      <session2>...</session2>
   </query>
</iq>
```

3.5.7 Receiving And Processing the session2 PDU

The initiator's user agent employs the following algorithm to process each session2 PDU:

- The attribute values are checked against the values sent in the session1 PDU. A mismatch results in an error code of 10008.

- The nonce is decoded, and its length is checked. The nonce may also be checked to detect replays. An invalid nonce results in an error code of 10002.

- The Diffie-Hellman group is checked against the value sent in the session1 PDU. A mismatch results in an error code of 10003

- The confidentiality and HMAC cryptographic algorithms are validated against the set proposed in the session1 PDU. A mismatch results in an error code of 10004.

- The authentication algorithm is validated against the set proposed in the session1 PDU. A mismatch results in an error code of 10005.

- The authenticator is verified. A failure results in an error code of 10007.

If any errors occur during processing, the session negotiation fails, and the error is communicated via the existing Jabber iq mechanism:

```
<iq from="initiator's␣JID" to="responder's␣JID" type="error" id="
    whatever">
   <error code="???">...</error>
</iq>
```

If no errors occur, then the initiator's user agent proceeds with the session3 PDU.

3.5.8 Generating And Sending the session3 PDU

The initiator's user agent employs the following algorithm to generate the session3 PDU:

- Appropriate values for the version, initiator, responder, sessionId, and hmac attributes are assembled. The version of this specification is '1.0'. The values of initiator and responder MUST be the JIDs of the two participants, respectively. The sessionId and hmac values MUST match the sessionId and hmac values contained in both the session1 and session2 PDUs.

- Based on the selected authentication algorithm, the initiator's authenticator is constructed. A digital signature algorithm requires calculating:
  - HK = hmac (initiator's nonce | responder's nonce, g^xy)
  - HASH_I = hmac (HK, version | sessionId | g^x | g^y | initiator's JID)

HASH_I is signed using the responder's private key and encoded in PKCS#1 format. The PKCS#1 octets are then further encoded in base64 or hex.
The passphrase algorithm requires calculating:

- HK = hmac (hash (passphrase), initiator's nonce | responder's nonce)

- HASH_I = hmac (HK, version | sessionId | g^x | g^y | initiator's JID)

The octets of HASH_I are simply encoded in base64 or hex.
The manner in which the initiator's user agent gains access to the initiator's credentials is outside the scope of this specification.

- A set of conversation keys may optionally be included in the response. This should typically be the case since security sessions are negotiated for the sole purpose of key transport.

These values are then used to prepare the XML session3 element; this element is transmitted via the existing Jabber iq mechanism:

```
<iq from="initiator's␣JID" to="responder's␣JID" type="result" id="
   whatever">
   <query xmlns="jabber:security:session">
      <session3>...</session3>
   </query>
</iq>
```

3.5.9 Receiving And Processing the session3 PDU

The responder's user agent employs the following algorithm to process each session3 PDU:

- The attribute values are checked against the values sent in the session2 PDU. A mismatch results in an error code of 10008.

- The authenticator is verified. A failure results in an error code of 10007.

- Any keys included in the PDU are processed and added to the user agent's key store.

If any errors occur during processing, the session negotiation fails, and the error is communicated via the existing Jabber iq mechanism:

```
<iq from="responder's␣JID" to="initiator's␣JID" type="error" id="
    whatever">
   <error code="???">...</error>
</iq>
```

3.5.10 Session Key Material Derivation

TBA

3.6 Key Transport

3.6.1 Overview

Conversation keys are used to protect conversation data.

3.6.2 The Key Transport Mechanism

Conversation keys are transported using the symmetric key wrap feature of XML Encryption embedded in the keyTransport PDU.

3.6.3 DTDs

```
<!ELEMENT keyTransport
          (convId, payload, hmac) >
<!ATTLIST keyTransport
          version CDATA #REQUIRED
          initiator CDATA #REQUIRED
          responder CDATA #REQUIRED
          sessionId CDATA #REQUIRED >

<!ELEMENT convId
          (#PCDATA)* >

<!-{}- These are actually instances of xenc:EncryptedKey -{}->
<!ELEMENT payload
          (confKey, hmacKey) >

<!ELEMENT hmac
          (#PCDATA)* >
<!ATTLIST hmac
          encoding (base64 | hex) #REQUIRED >
```

3.6.4 Generating And Sending the keyTransport PDU

The sender's user agent employs the following algorithm to generate the keyTransport PDU:

- Appropriate values for the version, initiator, responder, and sessionId attributes are assembled. The version of this specification is '1.0'. The values of initiator and responder MUST be the JIDs of the two participants who negotiated the security session, respectively, and they MUST correspond to an existing security session.
- The key's identifier, convId, is assembled.
- The payload, which consists of the confidentiality key and the integrity key, is wrapped in instances of xenc:EncryptedKey as follows:
  - The Type attribute of the xenc:EncryptedKey element MUST indicate 'content'.
  - The Id, MimeType and Encoding attributes of the xenc:EncryptedKey element MUST NOT be present.
  - The xenc:EncryptionMethod element MUST be present, and the Algorithm attribute MUST indicate a valid symmetric key wrap algorithm. Furthermore, the algorithm MUST be the same as was negotiated for the security session.
  - The ds:KeyInfo element MUST NOT be present. The key to use is SKc of the security session.
  - The xenc:CipherData element MUST be present, and it MUST use the CipherValue choice.
- The HMAC is computed using SKi of the security session over the following values:
  - the version attribute of the keyTransport element
  - the initiator attribute of the keyTransport element
  - the responder attribute of the keyTransport element
  - the sessionId attribute of the keyTransport element

  – the character string used to construct the body of the convId element

These values are then used to prepare the XML keyTransport element; this element is transmitted via the existing Jabber iq mechanism:

```
<iq from="sender's␣JID" to="receiver's␣JID" type="set" id="whatever">
   <query xmlns="jabber:security:keyTransport">
      <keyTransport>...</keyTransport>
   </query>
</iq>
```

3.6.5 Receiving and Processing the keyTransport PDU

The receiver's user agent employs the following algorithm to process each keyTransport PDU:

- The values of the version, initiator, responder, and sessionId are validated; initiator, responder, and sessionId MUST indicate an existing security session. A version mismatch results in an error code of 10000; an invalid security session results in an error of 10010.
- The payload, which consists of the confidentiality key and the intergrity key, is unwrapped. Any failures result in an error code of 10012.
- The body of the HMAC element is decoded into the actual HMAC octet string.
- The HMAC is validated. An invalid HMAC results in an error code of 10011.
- The keys are added to the user agent's key store.

If any errors occur during processing, the error is communicated via the existing Jabber iq mechanism:

```
<iq from="receiver's␣JID" to="sender's␣JID" type="error" id="whatever"
   >
   <error code="???">...</error>
</iq>
```

3.7 Message Protection

3.7.1 Overview

The ultimate goal is, of course, the protection of conversation data. The protocol exchanges described above allow the conversation participants to cryptographically protect their conversation data using the conversation keys that they share.

3.7.2 The Message Protection Mechanism

A protected message is defined as a traditional Jabber message whose body content is extended to include the transport of a cryptographically protected message body. The two key features are

- First, the usual body element contains some arbitrary text. Those familiar with the evolution of email protocols will recognize this trick as the same one used when MIME was introduced.
- Second, the message contains a Jabber x element defining the Jabber:security:message namespace; this element transports the protected message.

This mechanism has the advantages of allowing transparent integration with existing Jabber servers and existing Jabber clients.

3.7.3 DTD

```
<!ELEMENT protectedMessage
          (securityLabel?, payload, hmac) >
<!ATTLIST protectedMessage
          version CDATA #REQUIRED
          from CDATA #REQUIRED
          to CDATA #REQUIRED
          convId #REQUIRED
          seqNum #REQUIRED >

<!ELEMENT securityLabel
          (#PCDATA)* >

<!-{}- this is actually an instance of xenc:EncryptedData -{}->
<!ELEMENT payload
          (#PCDATA)* >

<!ELEMENT hmac
```

```
          (#PCDATA)* >
<!ATTLIST hmac
          encoding (base64 | hex) #REQUIRED >
```

3.7.4 Generating And Sending the protectedMessage PDU

The sender's user agent employs the following algorithm to generate the protectedMessage PDU:

- Appropriate values for the version, from, to, convId, and seqNum attributes are assembled. The version of this specification is '1.0'. The value of convId MUST correspond to an existing, valid key.

- The actual message body is encoded into a character string corresponding to a Jabber message body element. This character string is then wrapped in an instance of xenc:EncryptedData as follows:

  – The Type attribute of the xenc:EncryptedData element MUST indicate 'element'.

  – The Id, MimeType and Encoding attributes of the xenc:EncryptedData element MUST NOT be present.

  – The xenc:EncryptionMethod element MUST be present, and the Algorithm attribute MUST indicate a valid block encryption algorithm.

  – The ds:KeyInfo element MUST NOT be present. The key to be used is the confidentiality key indicated by the convId attribute.

  – The xenc:CipherData element MUST be present, and it MUST use the CipherValue choice.

- Using the HMAC key indicated by the convId attribute, the HMAC is computed over the following values:

  – the version attribute of the protectedMessage element

  – the from attribute of the protectedMessage element

  – the to attribute of the protectedMessage element

- the convId attribute of the protectedMessage element
- the seqNum attribute of the protectedMessage element
- any securityLabel element
- the character string used to construct the body of the payload element

These values are then used to prepare the XML protectedMessage element; this element is transmitted via the existing Jabber message mechanism:

```
<message from="sender's␣JID" to="reveiver's␣JID" id="whatever">
   <body>The real body is protected.</body>
   <x xmlns="jabber:security:message">
      <protectedMessage>...</protectedMessage>
   </x>
</iq>
```

3.7.5 Receiving and Processing the protectedMessage PDU

The receiver's user agent employs the following algorithm to process each protectedMessage PDU:

- The values of the version, from, to, convId, and seqNum are validated. A version mismatch results in an error code of 10000. An unknown convId results in an error code of 10015. If replay protection is utilized, a duplicate seqNum results in an error code of 10016.
- The body of the HMAC element is decoded into the actual HMAC octet string.
- The payload, which consists of the actual message body, is unwrapped. Any failures result in an error code of 10012.
- The HMAC is validated. An invalid HMAC results in an error code of 10011.

If any errors occur during processing, the error is communicated via the existing Jabber iq mechanism:

```
<iq from="receiver's␣JID" to="sender's␣JID" type="error" id="whatever"
    >
    <error code="???">...</error>
</iq>
```

3.8 Requesting Keys

TBA

3.9 Conformance Profile

The following block encryption algorithms are required, as specified by XML Encryption:

- http://www.w3.org/2001/04/xmlenc#tripledes-cbc
- http://www.w3.org/2001/04/xmlenc#aes128-cbc
- http://www.w3.org/2001/04/xmlenc#aes256-cbc

The following symmetric key wrap algorithms are required, as specified by XML Encryption:

- http://www.w3.org/2001/04/xmlenc#kw-tripledes
- http://www.w3.org/2001/04/xmlenc#kw-aes128
- http://www.w3.org/2001/04/xmlenc#kw-aes256

4 Diffie-Hellman Groups

This protocol makes use of the following Diffie-Hellman groups adopted from IKE.

4.1 1024 bit Group, modp1024

The hexidecimal value of the prime, p, is

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381
FFFFFFFF FFFFFFFF
```

The decimal value of the generator, g, is 2.

4.2 2048 bit Group, modp2048

The hexidecimal value of the prime, p, is

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
15728E5A 8AACAA68 FFFFFFFF FFFFFFFF
```

The decimal value of the generator, g, is 2.

4.3 4096 bit Group, modp4096

The hexidecimal value of the prime, p, is

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
15728E5A 8AAAC42D AD33170D 04507A33 A85521AB DF1CBA64
ECFB8504 58DBEF0A 8AEA7157 5D060C7D B3970F85 A6E1E4C7
ABF5AE8C DB0933D7 1E8C94E0 4A25619D CEE3D226 1AD2EE6B
F12FFA06 D98A0864 D8760273 3EC86A64 521F2B18 177B200C
```

```
BBE11757 7A615D6C 770988C0 BAD946E2 08E24FA0 74E5AB31
43DB5BFC E0FD108E 4B82D120 A9210801 1A723C12 A787E6D7
88719A10 BDBA5B26 99C32718 6AF4E23C 1A946834 B6150BDA
2583E9CA 2AD44CE8 DBBBC2DB 04DE8EF9 2E8EFC14 1FBECAA6
287C5947 4E6BC05D 99B2964F A090C3A2 233BA186 515BE7ED
1F612970 CEE2D7AF B81BDD76 2170481C D0069127 D5B05AA9
93B4EA98 8D8FDDC1 86FFB7DC 90A6C08F 4DF435C9 34063199
FFFFFFFF FFFFFFFF
```

The decimal value of the generator, g, is 2.

4.4 8192 bit Group, modp8192

The hexidecimal value of the prime, p, is

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
15728E5A 8AAAC42D AD33170D 04507A33 A85521AB DF1CBA64
ECFB8504 58DBEF0A 8AEA7157 5D060C7D B3970F85 A6E1E4C7
ABF5AE8C DB0933D7 1E8C94E0 4A25619D CEE3D226 1AD2EE6B
F12FFA06 D98A0864 D8760273 3EC86A64 521F2B18 177B200C
BBE11757 7A615D6C 770988C0 BAD946E2 08E24FA0 74E5AB31
43DB5BFC E0FD108E 4B82D120 A9210801 1A723C12 A787E6D7
88719A10 BDBA5B26 99C32718 6AF4E23C 1A946834 B6150BDA
2583E9CA 2AD44CE8 DBBBC2DB 04DE8EF9 2E8EFC14 1FBECAA6
287C5947 4E6BC05D 99B2964F A090C3A2 233BA186 515BE7ED
1F612970 CEE2D7AF B81BDD76 2170481C D0069127 D5B05AA9
93B4EA98 8D8FDDC1 86FFB7DC 90A6C08F 4DF435C9 34028492
36C3FAB4 D27C7026 C1D4DCB2 602646DE C9751E76 3DBA37BD
F8FF9406 AD9E530E E5DB382F 413001AE B06A53ED 9027D831
179727B0 865A8918 DA3EDBEB CF9B14ED 44CE6CBA CED4BB1B
DB7F1447 E6CC254B 33205151 2BD7AF42 6FB8F401 378CD2BF
5983CA01 C64B92EC F032EA15 D1721D03 F482D7CE 6E74FEF6
D55E702F 46980C82 B5A84031 900B1C9E 59E7C97F BEC7E8F3
23A97A7E 36CC88BE 0F1D45B7 FF585AC5 4BD407B2 2B4154AA
CC8F6D7E BF48E1D8 14CC5ED2 0F8037E0 A79715EE F29BE328
06A1D58B B7C5DA76 F550AA3D 8A1FBFF0 EB19CCB1 A313D55C
DA56C9EC 2EF29632 387FE8D7 6E3C0468 043E8F66 3F4860EE
12BF2D5B 0B7474D6 E694F91E 6DBE1159 74A3926F 12FEE5E4
```

```
38777CB6 A932DF8C D8BEC4D0 73B931BA 3BC832B6 8D9DD300
741FA7BF 8AFC47ED 2576F693 6BA42466 3AAB639C 5AE4F568
3423B474 2BF1C978 238F16CB E39D652D E3FDB8BE FC848AD9
22222E04 A4037C07 13EB57A8 1A23F0C7 3473FC64 6CEA306B
4BCBC886 2F8385DD FA9D4B7F A2C087E8 79683303 ED5BDD3A
062B3CF5 B3A278A6 6D2A13F8 3F44F82D DF310EE0 74AB6A36
4597E899 A0255DC1 64F31CC5 0846851D F9AB4819 5DED7EA1
B1D510BD 7EE74D73 FAF36BC3 1ECFA268 359046F4 EB879F92
4009438B 481C6CD7 889A002E D5EE382B C9190DA6 FC026E47
9558E447 5677E9AA 9E3050E2 765694DF C81F56E8 80B96E71
60C980DD 98EDD3DF FFFFFFFF FFFFFFFF
```

The decimal value of the generator, g, is 2.

5 Security Considerations

This entire document is about security.
This version of the protocol deliberately incorporates only a minimal amount of cryptographic choice. Examples of possible choices that can readily added in future drafts include:

- Support for the Digital Signature Standard
- Support for Elliptic Curve Cryptography
- Additional symmetric algorithms
- Additional hash algorithms

Furthermore, additional credential formats, such as OpenPGP, may be addressed in future drafts.
This version of the protocol includes a mechanism that derives a cryptographic key from a passphrase shared by a community of users. It is impossible to overstate the security issues that such a mechanism raises.
This version of the protocol does not include a specific rekeying capability. Data volumes in IM environments are expected to be small, and the protocol prefers to simply instantiate new conversation keys. It is straightforward to extend the security session protocol to enable negotiation of a new key.

6 Examples

6.1 Security Session

```
<iq from='initiator@some.tld'
    to='responder@other.tld'
    type='get'
    id='whatever' >
    <x xmlns='jabber:security:session>
␣␣␣␣␣␣␣<session1␣version='1.0'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣initiator='initiator@some.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣responder='responder@other.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sessionId='session11223344556677@some.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣hmac='hmac-sha1'>
␣␣␣␣␣␣␣␣␣<nonce␣encoding='hex'>
␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣</nonce>
␣␣␣␣␣␣␣␣␣<keyAgreement>
␣␣␣␣␣␣␣␣␣␣␣␣<dh␣group='modp4096'>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<publicKey␣encoding='base64'>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣</publicKey>
␣␣␣␣␣␣␣␣␣␣␣␣</dh>
␣␣␣␣␣␣␣␣␣</keyAgreement>
␣␣␣␣␣␣␣␣␣<algorithms>
␣␣␣␣␣␣␣␣␣␣␣␣<algorithm>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<confAlg␣cipher='3des-cbc'/>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<hmacAlg␣alg='hmac-sha1'/>
␣␣␣␣␣␣␣␣␣␣␣␣</algorithm>
␣␣␣␣␣␣␣␣␣</algorithms>
␣␣␣␣␣␣␣␣␣<authnMethods>
␣␣␣␣␣␣␣␣␣␣␣␣<authnMethod>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<digSig␣alg='rsa'>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<certificate␣type='x509'␣encoding='base64'>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣</certificate>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣</digSig>
␣␣␣␣␣␣␣␣␣␣␣␣</authnMethod>
␣␣␣␣␣␣␣␣␣</authnMethods>
␣␣␣␣␣␣</session1>
␣␣␣</x>
</iq>

<iq␣from='responder@other.tld'
␣␣␣␣to='initiator@some.tld'
␣␣␣␣type='result'
␣␣␣␣id='whatever'␣>
␣␣␣␣<x␣xmlns='jabber:security:session>
       <session2 version='1.0'
```

```
                 initiator='initiator@some.tld'
                 responder='responder@other.tld'
                 sessionId='session11223344556677@some.tld'
                 hmac='hmac-sha1'>
         <nonce encoding='hex'>
            ...
         </nonce>
         <keyAgreement>
            <dh group='modp4096'>
               <publicKey encoding='base64'>
                  ...
               </publicKey>
            </dh>
         </keyAgreement>
         <algorithm>
            <confAlg cipher='3des-cbc'/>
            <hmacAlg alg='hmac-sha1'/>
         </algorithm>
         <authnMethod>
            <digSig alg='rsa'>
               <certificate type='x509' encoding='base64'>
                  ...
               </certificate>
            </digSig>
         </authnMethod>
         <authenticator encoding='base64'>
            ...
         </authenticator>
      </session2>
   </x>
</iq>

<iq from='initiator@some.tld'
    to='responder@other.tld'
    type='result'
    id='whatever' >
    <x xmlns='jabber:security:session>
␣␣␣␣␣␣␣<session3␣version='1.0'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣initiator='initiator@some.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣responder='responder@other.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sessionId='session11223344556677@some.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣hmac='hmac-sha1'>
␣␣␣␣␣␣␣␣␣<authenticator␣encoding='base64'>
␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣</authenticator>
␣␣␣␣␣␣</session3>
␣␣␣</x>
</iq>
```

6.2 Key Transport

```
<iq type='set' >
   <x xmlns='jabber:security:key>
␣␣␣␣␣␣<keyTransport␣version='1.0'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣initiator='initiator@some.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣responder='responder@other.tld'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sessionId='session11223344556677@some.tld'>
␣␣␣␣␣␣␣␣␣<convId>
␣␣␣␣␣␣␣␣␣␣␣␣44d2d2d2d2@some.tld
␣␣␣␣␣␣␣␣␣</convId>
␣␣␣␣␣␣␣␣␣<EncryptedKey␣xmlns='http://www.w3.org/2001/04/xmlenc#'
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Type='http://www.w3.org/2001/04/xmlenc#Content
   '>
␣␣␣␣␣␣␣␣␣␣␣␣<EncryptionMethod␣Algorithm='http://www.w3.org/2001/04/
   xmlenc#kw-tripledes>
            </EncryptionMethod>
            <CipherData>
               <CipherValue>
                  ...
               </CipherValue>
            </CipherData>
         </EncryptedKey>
         <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'
                        Type='http://www.w3.org/2001/04/xmlenc#Content
                            '>
            <EncryptionMethod Algorithm='http://www.w3.org/2001/04/
               xmlenc#kw-tripledes>
␣␣␣␣␣␣␣␣␣␣␣␣</EncryptionMethod>
␣␣␣␣␣␣␣␣␣␣␣␣<CipherData>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<CipherValue>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣</CipherValue>
␣␣␣␣␣␣␣␣␣␣␣␣</CipherData>
␣␣␣␣␣␣␣␣␣</EncryptedKey>
␣␣␣␣␣␣␣␣␣<hmac␣encoding='hex'>
␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣</hmac>
␣␣␣␣␣␣</keyTransport>
␣␣␣</x>
</iq>
```

6.3 Message Protection

```
<message from='initiator@some.tld'
         to='responder@other.tld'>
   <body>
      The real body is protected.
   </body>
   <x xmlns='jabber:security:message'>
      <protectedMessage version='1.0'
                        from='initiator@some.tld'
                        to='responder@other.tld'
                        convId='44d2d2d2d2@some.tld'
                        seqNum='1'>
         <securityLabel>
            Confidential
         </securityLabel>
         <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
                        Type='http://www.w3.org/2001/04/xmlenc#Element
                           '>
            <EncryptionMethod Algorithm='http://www.w3.org/2001/04/
               xmlenc#tripledes-cbc>
␣␣␣␣␣␣␣␣␣␣␣␣</EncryptionMethod>
␣␣␣␣␣␣␣␣␣␣␣␣<CipherData>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<CipherValue>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣</CipherValue>
␣␣␣␣␣␣␣␣␣␣␣␣</CipherData>
␣␣␣␣␣␣␣␣␣</EncryptedData>
␣␣␣␣␣␣␣␣␣<hmac␣encoding='hex'>
␣␣␣␣␣␣␣␣␣␣␣␣...
␣␣␣␣␣␣␣␣␣</hmac>
␣␣␣␣␣␣</protectedMessage>
␣␣␣</x>
</message>
```

7 References

”XML Encryption Syntax and Processing”; http://www.w3.org/TR/xmlenc-core
more to be added