Off-the-Record Messaging Protocol version 2

This document describes version 2 of the Off-the-Record Messaging protocol. The main changes over version 1 include:

- Resolving the identity-binding flaw identified by Di Raimondo, Gennaro, and Krawczyk
- Not revealing the users' public keys to passive eavesdroppers; this could be useful if the application sending the OTR messages is also privacy-preserving
- Supporting fragmentation of OTR messages, to support IM networks whose maximum message size is very small.
- Adding a way to authenticate without the use of fingerprints.
- Better protocol version control, for future extensibility.

Very high level overview

OTR assumes a network model which provides in-order delivery of messages, but that some messages may not get delivered at all (for example, if the user disconnects). There may be an active attacker, who is allowed to perform a Denial of Service attack, but not to learn the contents of messages.

1. Alice signals to Bob that she would like (using an OTR Query Message) or is willing (using a whitespace-tagged plaintext message) to use OTR to communicate. Either mechanism should convey the version(s) of OTR that Alice is willing to use.
2. Bob initiates the authenticated key exchange (AKE) with Alice. Version 2 of OTR uses a variant of the SIGMA protocol as its AKE.
3. Alice and Bob exchange Data Messages to send information to each other.

High level overview

Requesting an OTR conversation

There are two ways Alice can inform Bob that she is willing to use the OTR protocol to speak with him: by sending him the OTR Query Message, or by including a special "tag" consisting of whitespace characters in one of her messages to him. Each method also includes a way for Alice to communicate to Bob which versions of the OTR protocol she is willing to speak with him.

The semantics of the OTR Query Message are that Alice is *requesting* that Bob start an OTR conversation with her (if, of course, he is willing and able to do so). On the other hand, the semantics of the whitespace tag are that Alice is merely *indicating* to Bob that she is willing and able to have an OTR conversation with him. If Bob has a policy of "only use OTR when it's explicitly requested", for example, then he *would* start an OTR conversation upon receiving an OTR Query Message, but *would not* upon receiving the whitespace tag.

Authenticated Key Exchange (AKE)

This section outlines the version of the SIGMA protocol used as the AKE. All exponentiations are done modulo a particular 1536-bit prime, and g is a generator of that group, as indicated in the detailed description below. Alice and Bob's long-term authentication public keys are pub_A and pub_B, respectively.

The general idea is that Alice and Bob do an *unauthenticated* Diffie-Hellman (D-H) key exchange to set up an encrypted channel, and then do mutual authentication *inside* that channel.

Bob will be initiating the AKE with Alice.

- Bob:
  1. Picks a random value r (128 bits)
  2. Picks a random value x (at least 320 bits)
  3. Sends Alice $AES_r(g^x)$, $HASH(g^x)$
- Alice:
  1. Picks a random value y (at least 320 bits)
  2. Sends Bob g^y
- Bob:
  1. Verifies that Alice's g^y is a legal value ($2 <= g^y <=$ modulus-2)
  2. Computes $s = (g^y)^x$
  3. Computes two AES keys c, c' and four MAC keys m1, m1', m2, m2' by hashing s in various ways
  4. Picks $keyid_B$, a serial number for his D-H key g^x
  5. Computes $M_B = MAC_{m1}(g^x, g^y, pub_B, keyid_B)$
  6. Computes $X_B = pub_B, keyid_B, sig_B(M_B)$
  7. Sends Alice r, $AES_c(X_B)$, $MAC_{m2}(AES_c(X_B))$
- Alice:
  1. Uses r to decrypt the value of g^x sent earlier
  2. Verifies that $HASH(g^x)$ matches the value sent earlier
  3. Verifies that Bob's g^x is a legal value ($2 <= g^x <=$ modulus-2)
  4. Computes $s = (g^x)^y$ (note that this will be the same as the value of s Bob calculated)
  5. Computes two AES keys c, c' and four MAC keys m1, m1', m2, m2' by hashing s in various ways (the same as Bob)
  6. Uses m2 to verify $MAC_{m2}(AES_c(X_B))$
  7. Uses c to decrypt $AES_c(X_B)$ to obtain $X_B = pub_B, keyid_B, sig_B(M_B)$
  8. Computes $M_B = MAC_{m1}(g^x, g^y, pub_B, keyid_B)$
  9. Uses pub_B to verify $sig_B(M_B)$
  10. Picks $keyid_A$, a serial number for her D-H key g^y
  11. Computes $M_A = MAC_{m1'}(g^y, g^x, pub_A, keyid_A)$
  12. Computes $X_A = pub_A, keyid_A, sig_A(M_A)$

13. Sends Bob $AES_{c'}(X_A)$, $MAC_{m2'}(AES_{c'}(X_A))$

- Bob:
  1. Uses m2' to verify $MAC_{m2'}(AES_{c'}(X_A))$
  2. Uses c' to decrypt $AES_{c'}(X_A)$ to obtain $X_A = pub_A, keyid_A, sig_A(M_A)$
  3. Computes $M_A = MAC_{m1'}(g^y, g^x, pub_A, keyid_A)$
  4. Uses pub_A to verify $sig_A(M_A)$
- If all of the verifications succeeded, Alice and Bob now know each other's Diffie-Hellman public keys, and share the value s. Alice is assured that s is known by someone with access to the private key corresponding to pub_B, and similarly for Bob.

Exchanging data

This section outlines the method used to protect data being exchanged between Alice and Bob. As above, all exponentiations are done modulo a particular 1536-bit prime, and g is a generator of that group, as indicated in the detailed description below.

Suppose Alice has a message (msg) to send to Bob.

- Alice:
  1. Picks the most recent of her own D-H encryption keys that Bob has acknowledged receiving (by using it in a Data Message, or failing that, in the AKE). Let key_A by that key, and let $keyid_A$ be its serial number.
  2. If the above key is Alice's most recent key, she generates a new D-H key (next_dh), to get the serial number $keyid_A+1$.
  3. Picks the most recent of Bob's D-H encryption keys that she has received from him (either in a Data Message or in the AKE). Let key_B by that key, and let $keyid_B$ be its serial number.
  4. Uses Diffie-Hellman to compute a shared secret from the two keys key_A and key_B, and generates the sending AES key, ek, and the sending MAC key, mk, as detailed below.
  5. Collects any old MAC keys that were used in previous messages, but will never again be used (because their associated D-H keys are no longer the most recent ones) into a list, oldmackeys.
  6. Picks a value of the counter, ctr, so that the triple (key_A, key_B, ctr) is never the same for more than one Data Message Alice sends to Bob.
  7. Computes $T_A = (keyid_A, keyid_B, next\_dh, ctr, AES\text{-}CTR_{ek,ctr}(msg))$
  8. Sends Bob T_A, $MAC_{mk}(T_A)$, oldmackeys
- Bob:
  1. Uses Diffie-Hellman to compute a shared secret from the two keys labelled by $keyid_A$ and $keyid_B$, and generates the receiving AES key, ek, and the receiving MAC key, mk, as detailed below. (These will be the same as the keys Alice generated, above.)
  2. Uses mk to verify $MAC_{mk}(T_A)$.
  3. Uses ek and ctr to decrypt $AES\text{-}CTR_{ek,ctr}(msg)$.

Socialist Millionaires' Protocol (SMP)

While data messages are being exchanged, either Alice or Bob may run SMP to detect impersonation or man-in-the-middle attacks. As above, all exponentiations are done modulo a particular 1536-bit prime, and g_1 is a generator of that group. All sent values include zero-knowledge proofs that they were generated according to this protocol, as indicated in the detailed description below. In the zero-knowledge proofs the D values are calculated modulo q = (p - 1) / 2, where p is the same 1536-bit prime as elsewhere. The random exponents are 1536-bit numbers.

Suppose Alice and Bob have secret information x and y respectively, and they wish to know whether x = y. The Socialist Millionaires' Protocol allows them to compare x and y without revealing any other information than the value of (x == y). For OTR, the secrets contain information about both parties' long-term authentication public keys, as well as information entered by the users themselves. If x = y, this means that Alice and Bob entered the same secret information, and so must be the same entities who established that secret to begin with.

Assuming that Alice begins the exchange:

- Alice:
  1. Picks random exponents a_2 and a_3
  2. Sends Bob $g_{2a} = g_1^{a_2}$ and $g_{3a} = g_1^{a_3}$
- Bob:
  1. Picks random exponents b_2 and b_3
  2. Computes $g_{2b} = g_1^{b_2}$ and $g_{3b} = g_1^{b_3}$
  3. Computes $g_2 = g_{2a}^{b_2}$ and $g_3 = g_{3a}^{b_3}$
  4. Picks random exponent r
  5. Computes $P_b = g_3^r$ and $Q_b = g_1^r\, g_2^y$
  6. Sends Alice g_{2b}, g_{3b}, P_b and Q_b
- Alice:
  1. Computes $g_2 = g_{2b}^{a_2}$ and $g_3 = g_{3b}^{a_3}$
  2. Picks random exponent s
  3. Computes $P_a = g_3^s$ and $Q_a = g_1^s\, g_2^x$
  4. Computes $R_a = (Q_a / Q_b)^{a_3}$
  5. Sends Bob P_a, Q_a and R_a
- Bob:
  1. Computes $R_b = (Q_a / Q_b)^{b_3}$
  2. Computes $R_{ab} = R_a^{b_3}$
  3. Checks whether $R_{ab} == (P_a / P_b)$
  4. Sends Alice R_b
- Alice:
  1. Computes $R_{ab} = R_b^{a_3}$
  2. Checks whether $R_{ab} == (P_a / P_b)$
- If everything is done correctly, then R_{ab} should hold the value of (P_a / P_b) times $(g_2^{a_3 b_3})^{(x - y)}$, which means that the test at the end of the protocol will only succeed if x == y. Further, since $g_2^{a_3 b_3}$ is a random number not known to any

party, if x is not equal to y, no other information is revealed.

Details of the protocol

Unencoded messages

This section describes the messages in the OTR protocol that are not base-64 encoded binary.

OTR Query Messages

If Alice wishes to communicate to Bob that she would like to use OTR, she sends a message containing the string "?OTR" followed by an indication of what versions of OTR she is willing to use with Bob. The version string is constructed as follows:

- If she is willing to use OTR version 1, the version string must start with "?".
- If she is willing to use OTR versions other than 1, a "v" followed by the byte identifiers for the versions in question, followed by "?". The byte identifier for OTR version 2 is "2". The order of the identifiers between the "v" and the "?" does not matter, but none should be listed more than once.

For example:

"?OTR?"
    Version 1 only
"?OTRv2?"
    Version 2 only
"?OTR?v2?"
    Versions 1 and 2
"?OTRv24x?"
    Version 2, and hypothetical future versions identified by "4" and "x"
"?OTR?v24x?"
    Versions 1, 2, and hypothetical future versions identified by "4" and "x"
"?OTR?v?"
    Also version 1 only
"?OTRv?"
    A bizarre claim that Alice would like to start an OTR conversation, but is unwilling to speak any version of the protocol

These strings may be hidden from the user (for example, in an attribute of an HTML tag), and/or may be accompanied by an explanitory message ("Alice has requested an Off-the-Record private conversation."). If Bob is willing to use OTR with Alice (with a protocol version that Alice has offered), he should start the AKE.

Tagged plaintext messages

If Alice wishes to communicate to Bob that she is willing to use OTR, she can attach a special whitespace tag to any plaintext message she sends him. This tag may occur anywhere in the message, and may be hidden from the user (as in the Query Messages, above).

The tag consists of the following 16 bytes, followed by one or more sets of 8 bytes indicating the version of OTR Alice is willing to use:

- Always send "\x20\x09\x20\x20\x09\x09\x09\x09" "\x20\x09\x20\x09\x20\x09\x20\x20", followed by one or more of:
- "\x20\x09\x20\x09\x20\x20\x09\x20" to indicate a willingness to use OTR version 1 with Bob (note: this string must come before all other whitespace version tags, if it is present, for backwards compatibility)
- "\x20\x20\x09\x09\x20\x20\x09\x20" to indicate a willingness to use OTR version 2 with Bob

If Bob is willing to use OTR with Alice (with a protocol version that Alice has offered), he should start the AKE. On the other hand, if Alice receives a plaintext message from Bob (rather than an initiation of the AKE), she should stop sending him the whitespace tag.

OTR Error Messages

Any message containing the string "?OTR Error:" is an OTR Error Message. The following part of the message should contain human-readable details of the error.

Encoded messages

This section describes the byte-level format of the base-64 encoded binary OTR messages. The binary form of each of the messages is described below. To transmit one of these messages, construct the ASCII string consisting of the five bytes "?OTR:", followed by the base-64 encoding of the binary form of the message, followed by the byte ".".

For the Diffie-Hellman group computations, the group is the one defined in RFC 3526 with 1536-bit modulus (hex, big-endian):

```
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 CA237327 FFFFFFFF FFFFFFFF
```

and a generator (g) of 2. Note that this means that whenever you see a Diffie-Hellman exponentiation in this document, it always means that the exponentiation is done modulo the above 1536-bit number.

Data types

Bytes (BYTE):
    1 byte unsigned value
Shorts (SHORT):
    2 byte unsigned value, big-endian
Ints (INT):
    4 byte unsigned value, big-endian

Multi-precision integers (MPI):
    4 byte unsigned len, big-endian
    len byte unsigned value, big-endian
    (MPIs must use the minimum-length encoding; i.e. no leading 0x00 bytes. This is important when calculating public key fingerprints.)
Opaque variable-length data (DATA):
    4 byte unsigned len, big-endian
    len byte data
Initial CTR-mode counter value (CTR):
    8 bytes data
Message Authentication Code (MAC):
    20 bytes MAC data

Public keys, signatures, and fingerprints

OTR users have long-lived public keys that they use for authentication (but *not* encryption). The current version of the OTR protocol only supports DSA public keys, but there is a key type marker for future extensibility.

OTR public authentication DSA key (PUBKEY):
    Pubkey type (SHORT)
        DSA public keys have type 0x0000
    p (MPI)
    q (MPI)
    g (MPI)
    y (MPI)
        (p,q,g,y) are the DSA public key parameters

OTR public keys are used to generate **signatures**; different types of keys produce signatures in different formats. The format for a signature made by a DSA public key is as follows:

DSA signature (SIG):
    (len is the length of the DSA public parameter q, which in current implementations must be 20 bytes, or 160 bits)
    len byte unsigned r, big-endian
    len byte unsigned s, big-endian

OTR public keys have **fingerprints**, which are hex strings that serve as identifiers for the public key. The fingerprint is calculated by taking the SHA-1 hash of the byte-level representation of the public key. However, there is an exception for backwards compatibility: if the pubkey type is 0x0000, those two leading 0x00 bytes are omitted from the data to be hashed. The encoding assures that, assuming the hash function itself has no useful collisions, and DSA keys have length less than 524281 bits (500 times larger than most DSA keys), no two public keys will have the same fingerprint.

D-H Commit Message

This is the first message of the AKE. Bob sends it to Alice to commit to a choice of D-H encryption key (but the key itself is not yet revealed). This allows the secure session id to be much shorter than in OTR version 1, while still preventing a man-in-the-middle attack on it.

Protocol version (SHORT)
The version number of this protocol is 0x0002.
Message type (BYTE)
The D-H Commit Message has type 0x02.
Encrypted g^x (DATA)
Produce this field as follows:

- Choose a random value r (128 bits)
- Choose a random value x (at least 320 bits)
- Serialize g^x as an MPI, gxmpi. [gxmpi will probably be 196 bytes long, starting with "\x00\x00\x00\xc0".]
- Encrypt gxmpi using AES128-CTR, with key r and initial counter value 0. The result will be the same length as gxmpi.
- Encode this encrypted value as the DATA field.

Hashed g^x (DATA)
This is the SHA256 hash of gxmpi.

D-H Key Message

This is the second message of the AKE. Alice sends it to Bob, and it simply consists of Alice's D-H encryption key.

Protocol version (SHORT)
The version number of this protocol is 0x0002.
Message type (BYTE)
The D-H Key Message has type 0x0a.
g^y (MPI)
Choose a random value y (at least 320 bits), and calculate g^y.

Reveal Signature Message

This is the third message of the AKE. Bob sends it to Alice, revealing his D-H encryption key (and thus opening an encrypted channel), and also authenticating himself (and the parameters of the channel, preventing a man-in-the-middle attack on the channel itself) to Alice.

Protocol version (SHORT)
The version number of this protocol is 0x0002.
Message type (BYTE)
The Reveal Signature Message has type 0x11.
Revealed key (DATA)
This is the value r picked earlier.
Encrypted signature (DATA)
This field is calculated as follows:

- Compute the Diffie-Hellman shared secret s.
- Use s to compute an AES key c and two MAC keys m1 and m2, as specified below.
- Select $keyid_B$, a serial number for the D-H key computed earlier. It is an

INT, and must be greater than 0.

- Compute the 32-byte value M_B to be the SHA256-HMAC of the following data, using the key m1:

  g^x (MPI)
  g^y (MPI)
  pub_B (PUBKEY)
  $keyid_B$ (INT)
- Let X_B be the following structure:

  pub_B (PUBKEY)
  $keyid_B$ (INT)
  $sig_B(M_B)$ (SIG)

  This is the signature, using the private part of the key pub_B, of the 32-byte M_B (which does not need to be hashed again to produce the signature).
- Encrypt X_B using AES128-CTR with key c and initial counter value 0.
- Encode this encrypted value as the DATA field.

MAC'd signature (MAC)

This is the SHA256-HMAC-160 (that is, the first 160 bits of the SHA256-HMAC) of the encrypted signature field (including the four-byte length), using the key m2.

Signature Message

This is the final message of the AKE. Alice sends it to Bob, authenticating herself and the channel parameters to him.

Protocol version (SHORT)

The version number of this protocol is 0x0002.

Message type (BYTE)

The Signature Message has type 0x12.

Encrypted signature (DATA)

This field is calculated as follows:

- Compute the Diffie-Hellman shared secret s.
- Use s to compute an AES key c' and two MAC keys m1' and m2', as specified below.
- Select $keyid_A$, a serial number for the D-H key computed earlier. It is an INT, and must be greater than 0.
- Compute the 32-byte value M_A to be the SHA256-HMAC of the following data, using the key m1':

  g^y (MPI)
  g^x (MPI)
  pub_A (PUBKEY)
  $keyid_A$ (INT)
- Let X_A be the following structure:

  pub_A (PUBKEY)
  $keyid_A$ (INT)
  $sig_A(M_A)$ (SIG)

This is the signature, using the private part of the key pub_A, of the 32-byte M_A (which does not need to be hashed again to produce the signature).

- Encrypt X_A using AES128-CTR with key c' and initial counter value 0.
- Encode this encrypted value as the DATA field.

MAC'd signature (MAC)

This is the SHA256-HMAC-160 (that is, the first 160 bits of the SHA256-HMAC) of the encrypted signature field (including the four-byte length), using the key m2'.

Data Message

This message is used to transmit a private message to the correspondent. It is also used to reveal old MAC keys.

The plaintext message (either before encryption, or after decryption) consists of a human-readable message (encoded in UTF-8, optionally with HTML markup), optionally followed by:

- a single NUL (a BYTE with value 0x00), **and**
- zero or more TLV (type/length/value) records (with no padding between them)

Each TLV record is of the form:

Type (SHORT)

The type of this record. Records with unrecognized types should be ignored.

Length (SHORT)

The length of the following field

Value (len BYTEs) [where len is the value of the Length field]

Any pertinent data for the record type.

Some TLV examples:

\x00\x01\x00\x00

A TLV of type 1, containing no data

\x00\x00\x00\x05\x68\x65\x6c\x6c\x6f

A TLV of type 0, containing the value "hello"

The currently defined TLV record types are:

Type 0: Padding

The value may be an arbitrary amount of data, which should be ignored. This type can be used to disguise the length of the plaintext message.

Type 1: Disconnected

If the user requests to close the private connection, you may send a message (possibly with empty human-readable part) containing a record with this TLV type just before you discard the session keys, and transition to MSGSTATE_PLAINTEXT (see below). If you receive a TLV record of this type, you should transition to MSGSTATE_FINISHED (see below), and inform the user that his correspondent has closed his end of the private connection, and the user should do the same.

Type 2: SMP Message 1
: The value represents an initiating message of the Socialist Millionaires' Protocol, described below.

Type 3: SMP Message 2
: The value represents the second message in an instance of SMP.

Type 4: SMP Message 3
: The value represents the third message in an instance of SMP.

Type 5: SMP Message 4
: The value represents the final message in an instance of SMP.

Type 6: SMP Abort Message
: If the user cancels SMP prematurely or encounters an error in the protocol and cannot continue, you may send a message (possibly with empty human-readable part) with this TLV type to instruct the other party's client to abort the protocol. The associated length should be zero and the associated value should be empty. If you receive a TLV of this type, you should change the SMP state to SMP_EXPECT1 (see below).

SMP Message TLVs (types 2-5) all carry data sharing the same general format:

MPI count (INT)
: The number of MPIs contained in the remainder of the TLV.

MPI 1 (MPI)
: The first MPI of the TLV, serialized into a byte array.

MPI 2 (MPI)
: The second MPI of the TLV, serialized into a byte array.

etc.

There should be as many MPIs as declared in the MPI count field. For the exact MPIs passed for each SMP TLV, see the SMP state machine below.

A message with an empty human-readable part (the plaintext is of zero length, or starts with a NUL) is a "heartbeat" packet, and should not be displayed to the user. (But it's still useful to effect key rotations.)

Data Message format:

Protocol version (SHORT)
: The version number of this protocol is 0x0002.

Message type (BYTE)
: The Data Message has type 0x03.

Flags (BYTE)
: The bitwise-OR of the flags for this message. Usually you should set this to 0x00. The only currently defined flag is:

  IGNORE_UNREADABLE (0x01)
  : If you receive a Data Message with this flag set, and you are unable to decrypt the message or verify the MAC (because, for example, you don't have the right keys), just ignore the message instead of producing some kind of error or notification to the user.

Sender keyid (INT)
: Must be strictly greater than 0, and increment by 1 with each key change

Recipient keyid (INT)
: Must therefore be strictly greater than 0, as the receiver has no key with id 0.

The sender and recipient keyids are those used to encrypt and MAC this message.

DH y (MPI)

The *next* [i.e. sender_keyid+1] public key for the sender

Top half of counter init (CTR)

This should monotonically increase (as a big-endian value) for each message sent with the same (sender keyid, recipient keyid) pair, and must not be all 0x00.

Encrypted message (DATA)

Using the appropriate encryption key (see below) derived from the sender's and recipient's DH public keys (with the keyids given in this message), perform AES128 counter-mode (CTR) encryption of the message. The initial counter is a 16-byte value whose first 8 bytes are the above "top half of counter init" value, and whose last 8 bytes are all 0x00. Note that counter mode does not change the length of the message, so no message padding needs to be done. If you *want* to do message padding (to disguise the length of your message), use the above TLV of type 0.

Authenticator (MAC)

The SHA1-HMAC, using the appropriate MAC key (see below) of everything from the Protocol version to the end of the encrypted message

Old MAC keys to be revealed (DATA)

See "Revealing MAC Keys", below.

Socialist Millionaires' Protocol (SMP)

The Socialist Millionaires' Protocol allows two parties with secret information x and y respectively to check whether (x==y) without revealing any additional information about the secrets. The protocol used by OTR is based on the work of Boudot, Schoenmakers and Traore (2001). A full justification for its use in OTR is made by Alexander and Goldberg, in a paper published in 2007. The following is a technical account of what is transmitted during the course of the protocol.

Secret information

The secret information x and y compared during this protocol contains not only information entered by the users, but also information unique to the conversation in which SMP takes place. Specifically, the format is:

Version (BYTE)

The version of SMP used. The version described here is 1.

Initiator fingerprint (20 BYTEs)

The fingerprint that the party initiating SMP is using in the current conversation.

Responder fingerprint (20 BYTEs)

The fingerprint that the party that did not initiate SMP is using in the current conversation.

Secure Session ID

The ssid described below.

User input

The input string given by the user at runtime.

Then the SHA256 hash of the above is taken, and the digest becomes the actual secret (x or y) to be used in SMP. The additional fields insure that not only do both parties know the same secret input string, but no man-in-the-middle is capable of reading their communication either.

The SMP state machine

Whenever the OTR message state machine has MSGSTATE_ENCRYPTED set (see below), the SMP state machine may progress. If at any point MSGSTATE_ENCRYPTED becomes unset, SMP must abandon its state and return to its initial setup. The SMP state consists of one main variable, as well as information from the partial computations at each protocol step.

Expected Message

This main state variable for SMP controls what SMP-specific TLVs will be accepted. This variable has no effect on type 0 or type 1 TLVs, which are always allowed. smpstate can take one of four values:

SMPSTATE_EXPECT1
: This state indicates that only type 2 TLVs (SMP message 1) should be accepted. This is the default state when SMP has not yet begun. This state is also reached whenever an error occurs or SMP is aborted, and the protocol must be restarted from the beginning.

SMPSTATE_EXPECT2
: This state indicates that only type 3 TLVs (SMP message 2) should be accepted.

SMPSTATE_EXPECT3
: This state indicates that only type 4 TLVs (SMP message 3) should be accepted.

SMPSTATE_EXPECT4
: This state indicates that only type 5 TLVs (SMP message 4) should be accepted.

State Transitions

There are 7 actions that an OTR client must handle:

- Received TLVs:
  - SMP Message 1
  - SMP Message 2
  - SMP Message 3
  - SMP Message 4
  - SMP Abort Message
- User actions:
  - User requests to begin SMP
  - User requests to abort SMP

The following sections outline what is to be done in each case. They all assume that MSGSTATE_ENCRYPTED is set. For simplicity, they also assume that Alice has begun SMP, and Bob is responding to her.

SMP Hash function

In the following actions, there are many places where a SHA256 hash of an integer followed by one or two MPIs is taken. The input to this hash function is:

Version (BYTE)
: This distinguishes calls to the hash function at different points in the protocol, to prevent Alice from replaying Bob's zero knowledge proofs or vice versa.

First MPI (MPI)
: The first MPI given as input, serialized in the usual way.

Second MPI (MPI)
: The second MPI given as input, if present, serialized in the usual way. If only one MPI is given as input, this field is simply omitted.

Receiving a type 2 TLV (SMP message 1)

SMP message 1 is sent by Alice to begin a DH exchange to determine two new generators, g_2 and g_3. It contains the following mpi values:

g_{2a}
: Alice's half of the DH exchange to determine g_2.

c2, D2
: A zero-knowledge proof that Alice knows the exponent associated with her transmitted value g_{2a}.

g_{3a}
: Alice's half of the DH exchange to determine g_3.

c3, D3
: A zero-knowledge proof that Alice knows the exponent associated with her transmitted value g_{3a}.

When Bob receives this TLV he should do:

If smpstate is not SMPSTATE_EXPECT1:
: Set smpstate to SMPSTATE_EXPECT1 and send a type 6 TLV (SMP abort) to Alice.

If smpstate is SMPSTATE_EXPECT1:
: Verify Alice's zero-knowledge proofs for g_{2a} and g_{3a}:

  1. Check that $c2 = SHA256(1, g_1^{D2} g_{2a}^{c2})$.
  2. Check that $c3 = SHA256(2, g_1^{D3} g_{3a}^{c3})$.

  Create a type 3 TLV (SMP message 2) and send it to Alice:

  1. Determine Bob's secret input y, which is to be compared to Alice's secret x.
  2. Pick random exponents b_2 and b_3. These will used during the DH exchange to pick generators.
  3. Pick random exponents r2, r3, r4, r5 and r6. These will be used to add a blinding factor to the final results, and to generate zero-knowledge proofs that this message was created honestly.
  4. Compute $g_{2b} = g_1^{b_2}$ and $g_{3b} = g_1^{b_3}$
  5. Generate a zero-knowledge proof that the exponent b_2 is known by setting $c2 = SHA256(3, g_1^{r2})$ and $D2 = r2 - b_2\ c2 \bmod q$.
  6. Generate a zero-knowledge proof that the exponent b_3 is known by setting $c3 = SHA256(4, g_1^{r3})$ and $D3 = r3 - b_3\ c3 \bmod q$.

7. Compute $g_2 = g_{2a}^{b_2}$ and $g_3 = g_{3a}^{b_3}$
8. Compute $P_b = g_3^{r4}$ and $Q_b = g_1^{r4} g_2^{y}$
9. Generate a zero-knowledge proof that P_b and Q_b were created according to the protocol by setting cP = SHA256(5, g_3^{r5}, $g_1^{r5} g_2^{r6}$), D5 = r5 - r4 cP mod q and D6 = r6 - y cP mod q.
10. Store the values of g_{3a}, g_2, g_3, b_3, P_b and Q_b for use later in the protocol.
11. Send Alice a type 3 TLV (SMP message 2) containing g_{2b}, c2, D2, g_{3b}, c3, D3, P_b, Q_b, cP, D5 and D6, in that order.

Set smpstate to SMPSTATE_EXPECT3.

Receiving a type 3 TLV (SMP message 2)

SMP message 2 is sent by Bob to complete the DH exchange to determine the new generators, g_2 and g_3. It also begins the construction of the values used in the final comparison of the protocol. It contains the following mpi values:

g_{2b}

Bob's half of the DH exchange to determine g_2.

c2, D2

A zero-knowledge proof that Bob knows the exponent associated with his transmitted value g_{2b}.

g_{3b}

Bob's half of the DH exchange to determine g_3.

c3, D3

A zero-knowledge proof that Bob knows the exponent associated with his transmitted value g_{3b}.

P_b, Q_b

These values are used in the final comparison to determine if Alice and Bob share the same secret.

cP, D5, D6

A zero-knowledge proof that P_b and Q_b were created according to the protcol given above.

When Alice receives this TLV she should do:

If smpstate is not SMPSTATE_EXPECT2:

Set smpstate to SMPSTATE_EXPECT1 and send a type 6 TLV (SMP abort) to Bob.

If smpstate is SMPSTATE_EXPECT2:

Verify Bob's zero-knowledge proofs for g_{2b}, g_{3b}, P_b and Q_b:

1. Check that c2 = SHA256(3, $g_1^{D2} g_{2b}^{c2}$).
2. Check that c3 = SHA256(4, $g_1^{D3} g_{3b}^{c3}$).
3. Check that cP = SHA256(5, $g_3^{D5} P_b^{cP}$, $g_1^{D5} g_2^{D6} Q_b^{cP}$).

Create a type 4 TLV (SMP message 3) and send it to Bob:

1. Pick random exponents r4, r5, r6 and r7. These will be used to add a blinding factor to the final results, and to generate zero-knowledge proofs that this message was created honestly.

2. Compute $g_2 = g_{2b}^{a_2}$ and $g_3 = g_{3b}^{a_3}$
3. Compute $P_a = g_3^{r4}$ and $Q_a = g_1^{r4} g_2^{x}$
4. Generate a zero-knowledge proof that P_a and Q_a were created according to the protocol by setting cP = SHA256(6, g_3^{r5}, $g_1^{r5} g_2^{r6}$), D5 = r5 - r4 cP mod q and D6 = r6 - x cP mod q.
5. Compute $R_a = (Q_a / Q_b)^{a_3}$
6. Generate a zero-knowledge proof that R_a was created according to the protocol by setting cR = SHA256(7, g_1^{r7}, $(Q_a / Q_b)^{r7}$) and D7 = r7 - a_3 cR mod q.
7. Store the values of g_{3b}, (P_a / P_b), (Q_a / Q_b) and R_a for use later in the protocol.
8. Send Bob a type 4 TLV (SMP message 3) containing P_a, Q_a, cP, D5, D6, R_a, cR and D7 in that order.

Set smpstate to SMPSTATE_EXPECT4.

Receiving a type 4 TLV (SMP message 3)

SMP message 3 is Alice's final message in the SMP exchange. It has the last of the information required by Bob to determine if x = y. It contains the following mpi values:

P_a, Q_a
: These values are used in the final comparison to determine if Alice and Bob share the same secret.

cP, D5, D6
: A zero-knowledge proof that P_a and Q_a were created according to the protcol given above.

R_a
: This value is used in the final comparison to determine if Alice and Bob share the same secret.

cR, D7
: A zero-knowledge proof that R_a was created according to the protcol given above.

When Bob receives this TLV he should do:

If smpstate is not SMPSTATE_EXPECT3:
    Set smpstate to SMPSTATE_EXPECT1 and send a type 6 TLV (SMP abort) to Bob.
If smpstate is SMPSTATE_EXPECT3:
    Verify Alice's zero-knowledge proofs for P_a, Q_a and R_a:

1. Check that cP = SHA256(6, $g_3^{D5} P_a^{cP}$, $g_1^{D5} g_2^{D6} Q_a^{cP}$).
2. Check that cR = SHA256(7, $g_1^{D7} g_{3a}^{cR}$, $(Q_a / Q_b)^{D7} R_a^{cR}$).

Create a type 5 TLV (SMP message 4) and send it to Alice:

1. Pick a random exponent r7. This will be used to generate Bob's final zero-knowledge proof that this message was created honestly.
2. Compute $R_b = (Q_a / Q_b)^{b_3}$

3. Generate a zero-knowledge proof that R_b was created according to the protocol by setting cR = SHA256(8, g_1^{r7}, $(Q_a / Q_b)^{r7}$) and D7 = r7 - b_3 cR mod q.
4. Send Alice a type 5 TLV (SMP message 4) containing R_b, cR and D7 in that order.

Check whether the protocol was successful:

1. Compute $R_{ab} = R_a^{b_3}$.
2. Determine if x = y by checking the equivalent condition that $(P_a / P_b) = R_{ab}$.

Set smpstate to SMPSTATE_EXPECT1, as no more messages are expected from Alice.

Receiving a type 5 TLV (SMP message 4)

SMP message 4 is Bob's final message in the SMP exchange. It has the last of the information required by Alice to determine if x = y. It contains the following mpi values:

R_b
: This value is used in the final comparison to determine if Alice and Bob share the same secret.

cR, D7
: A zero-knowledge proof that R_b was created according to the protcol given above.

When Alice receives this TLV she should do:

If smpstate is not SMPSTATE_EXPECT4:
: Set smpstate to SMPSTATE_EXPECT1 and send a type 6 TLV (SMP abort) to Bob.

If smpstate is SMPSTATE_EXPECT4:
: Verify Bob's zero-knowledge proof for R_b:

1. Check that cR = SHA256(8, $g_1^{D7} g_{3b}^{cR}$, $(Q_a / Q_b)^{D7} R_b^{cR}$).

Check whether the protocol was successful:

1. Compute $R_{ab} = R_b^{a_3}$.
2. Determine if x = y by checking the equivalent condition that $(P_a / P_b) = R_{ab}$.

Set smpstate to SMPSTATE_EXPECT1, as no more messages are expected from Bob.

User requests to begin SMP

If smpstate is not set to SMPSTATE_EXPECT1:
: SMP is already underway. If you wish to restart SMP, send a type 6 TLV (SMP abort) to the other party and then proceed as if smpstate was SMPSTATE_EXPECT1. Otherwise, you may simply continue the current SMP instance.

If smpstate is set to SMPSTATE_EXPECT1:
: No current exchange is underway. In this case, Alice should create a valid type 2 TLV (SMP message 1) as follows:

1. Determine her secret input x, which is to be compared to Bob's secret y.

2. Pick random values a_2 and a_3 (128 bits). These will be Alice's exponents for the DH exchange to pick generators.
3. Pick random values r2 and r3 (128 bits). These will be used to generate zero-knowledge proofs that this message was created according to the protocol.
4. Compute $g_{2a} = g_1^{a_2}$ and $g_{3a} = g_1^{a_3}$
5. Generate a zero-knowledge proof that the exponent a_2 is known by setting $c2 = SHA256(1, g_1^{r2})$ and $D2 = r2 - a_2\ c2 \bmod q$.
6. Generate a zero-knowledge proof that the exponent a_3 is known by setting $c3 = SHA256(2, g_1^{r3})$ and $D3 = r3 - a_3\ c3 \bmod q$.
7. Store the values of x, a_2 and a_3 for use later in the protocol.
8. Send Bob a type 2 TLV (SMP message 1) containing g_{2a}, c2, D2, g_{3a}, c3 and D3 in that order.

Set smpstate to SMPSTATE_EXPECT2.

User requests to abort SMP

In all cases, send a type 6 TLV (SMP abort) to the correspondent and set smpstate to SMPSTATE_EXPECT1.

Key Management

For each correspondent, keep track of:

Your two most recent DH public/private key pairs
 our_dh[our_keyid] (most recent) and our_dh[our_keyid-1] (previous)
His two most recent DH public keys
 their_y[their_keyid] (most recent) and their_y[their_keyid-1] (previous)

When starting a private conversation with a correspondent, generate two DH key pairs for yourself, and set our_keyid = 2. Note that all DH key pairs should have a private part that is at least 320 bits long.

When you send AKE messages:
 Send the public part of our_dh[our_keyid-1], with the keyid field, of course, set to (our_keyid-1).

Upon completing the AKE:
 If the specified keyid equals either their_keyid or their_keyid-1, and the DH pubkey contained in the AKE messages matches the one you've stored for that keyid, that's great. Otherwise, forget all values of their_y[], and of their_keyid, and set their_keyid to the keyid value given in the AKE messages, and their_y[their_keyid] to the DH pubkey value given in the AKE messages. their_y[their_keyid-1] should be set to NULL.

When you send a Data Message:
 Set the sender keyid to (our_keyid-1), and the recipient keyid to (their_keyid). Set the DH pubkey in the Data message to the public part of our_dh[our_keyid]. Use our_dh[our_keyid-1] and their_y[their_keyid] to calculate session keys, as

outlined below. Use the "sending AES key" to encrypt the message, and the "sending MAC key" to calculate its MAC.

When you receive a Data Message:

Use the keyids in the message to select which of your DH key pairs and which of his DH pubkeys to use to verify the MAC. If the keyids do not represent either the most recent key or the previous key (for either the sender or receiver), reject the message. Also reject the message if the sender keyid is their_keyid-1, but their_y[their_keyid-1] is NULL.

Otherwise, calculate the session keys as outlined below. Use the "receiving MAC key" to verify the MAC on the message. If it does not verify, reject the message.

Check that the counter in the Data message is strictly larger than the last counter you saw using this pair of keys. If not, reject the message.

If the MAC verifies, decrypt the message using the "receiving AES key".

Finally, check if keys need rotation:

- If the "recipient keyid" in the Data message equals our_keyid, then he's seen the public part of our most recent DH key pair, so you must securely forget our_dh[our_keyid-1], increment our_keyid, and set our_dh[our_keyid] to a new DH key pair which you generate.
- If the "sender keyid" in the Data message equals their_keyid, increment their_keyid, and set their_y[their_keyid] to the new DH pubkey specified in the Data message.

Computing AES keys, MAC keys, and the secure session id

OTR uses Diffie-Hellman to calculate shared secrets in the usual way: if Bob knows x, and tells Alice g^x, and Alice knows y, and tells Bob g^y, then they each can calculate s = g^{xy}: Alice calculates $(g^x)^y$, and Bob calculates $(g^y)^x$.

During the AKE, Alice and Bob each calculate s in this way, and then they each compute seven values based on s:

- A 64-bit secure session id, ssid
- Two 128-bit AES encryption keys, c and c'
- Four 256-bit SHA256-HMAC keys, m1, m2, m1', and m2'

This is done in the following way:

- Write the value of s as a minimum-length MPI, as specified above (4-byte big-endian len, len-byte big-endian value). Let this (4+len)-byte value be "secbytes".
- For a given byte b, define h2(b) to be the 256-bit output of the SHA256 hash of the (5+len) bytes consisting of the byte b followed by secbytes.
- Let ssid be the first 64 bits of h2(0x00).
- Let c be the first 128 bits of h2(0x01), and let c' be the second 128 bits of h2(0x01).

- Let m1 be h2(0x02).
- Let m2 be h2(0x03).
- Let m1' be h2(0x04).
- Let m2' be h2(0x05).

c, m1, and m2 are used to create and verify the Reveal Signature Message; c', m1', and m2' are used to create and verify the Signature message.

If the user requests to see the secure session id, it should be displayed as two 32-bit bigendian unsigned values, in C "%08x" format. If the user transmitted the Reveal Signature message during the AKE that produced this ssid, then display the first 32 bits in bold, and the second 32 bits in non-bold. If the user transmitted the Signature message instead, display the first 32 bits in non-bold, and the second 32 bits in bold. This session id can be used by the parties to verify (say, over the telephone, assuming the parties recognize each others' voices) that there is no man-in-the-middle by having each side read his bold part to the other. [Note that this only needs to be done in the event that the users do not trust that their long-term signature keys have not been compromised.]

During the exchange of Data Messages, Alice and Bob use the keyids listed in the Data Message to select Diffie-Hellman keys to use to compute s, and the (4+len)-byte value of secbytes, as above.

From this, they calculate four values:

- Two 128-bit AES encryption keys, the "sending AES key", and the "receiving AES key"
- Two 160-bit SHA1-HMAC keys, the "sending MAC key", and the "receiving MAC key"

These keys are calculated as follows:

- Alice (and similarly for Bob) determines if she is the "low" end or the "high" end of this Data Message. If Alice's public key is numerically greater than Bob's public key, then she is the "high" end. Otherwise, she is the "low" end. Note that who is the "low" end and who is the "high" end can change every time a new D-H public key is exchanged in a Data Message.
- She sets the values of "sendbyte" and "recvbyte" according to whether she is the the "low" or the "high" end of the Data Message:
    - If she is the "high" end, she sets "sendbyte" to 0x01 and "recvbyte" to 0x02.
    - If she is the "low" end, she sets "sendbyte" to 0x02 and "recvbyte" to 0x01.
- For a given byte b, define h1(b) to be the 160-bit output of the SHA-1 hash of the (5+len) bytes consisting of the byte b, followed by secbytes.
- The "sending AES key" is the first 16 bytes of h1(sendbyte).
- The "sending MAC key" is the 20-byte SHA-1 hash of the 16-byte sending AES key.
- The "receiving AES key" is the first 16 bytes of h1(recvbyte).
- The "receiving MAC key" is the 20-byte SHA-1 hash of the 16-byte receiving AES key.

Revealing MAC keys

Whenever you are about to forget either one of your old D-H key pairs, or one of your correspondent's old D-H public keys, take all of the receiving MAC keys that were generated by that key (note that there are up to two: the receiving MAC keys produced by the pairings of that key with each of two of the other side's keys; but note that you only need to take MAC keys that were actually used to verify a MAC on a message), and put them (as a set of concatenated 20-byte values) into the "Old MAC keys to be revealed" section of the next Data Message you send. This in done to allow the forgeability of OTR transcripts: once the MAC keys are revealed, anyone can modify an OTR message and still have it appear valid. But since we don't reveal the MAC keys until their corresponding pubkeys are being discarded, there is no danger of accepting a message as valid which uses a MAC key which has already been revealed.

Fragmentation

Some networks may have a maximum message size that is too small to contain an encoded OTR message. In that event, the sender may choose to split the message into a number of *fragments*. This section describes the format of the fragments. All OTR version 2 clients must be able to assemble received fragments, but performing fragmentation on outgoing messages is optional.

Transmitting Fragments

If you have information about the maximum size of message you are able to send (the different IM networks have different limits), you can fragment an encoded OTR message as follows:

- Start with the OTR message as you would normally transmit it. For example, a Data Message would start with "?OTR:AAED" and end with ".".
- Break it up into sufficiently small pieces. Let the number of pieces be (n), and the pieces be piece[1],piece[2],...,piece[n].
- Transmit (n) messages with the following (printf-like) structure (as k runs from 1 to n inclusive):

  "?OTR,%hu,%hu,%s," , k , n , piece[k]
- Note that k and n are unsigned short ints (2 bytes), and each has a maximum value of 65535. Also, each piece[k] must be non-empty.

Receiving Fragments:

If you receive a message containing "?OTR," (note that you'll need to check for this _before_ checking for any of the other "?OTR:" markers):

- Parse it as the printf statement above into k, n, and piece.
- Let (K,N) be your currently stored fragment number, and F be your currently stored fragment. [If you have no currently stored fragment, then K = N = 0 and F = "".]
- If k == 0 or n == 0 or k > n, discard this (illegal) fragment.
- If k == 1:
  - Forget any stored fragment you may have
  - Store (piece) as F.
  - Store (k,n) as (K,N).
- If n == N and k == K+1:
  - Append (piece) to F.

    - Store (k,n) as (K,N).
- Otherwise:
    - Forget any stored fragment you may have
    - Store "" as F.
    - Store (0,0) as (K,N).

After this, if N > 0 and K == N, treat F as the received message.

If you receive a non-OTR message, or an unfragmented message, forget any stored fragment you may have, store "" as F and store (0,0) as (K,N).

For example, here is a Data Message we would like to transmit over a network with an unreasonably small maximum message size:

```
?OTR:AAEDAAAAAQAAAAEAAADAVf3Ei72ZgFeKqWvLMnuVPVCwxktsOZ1Qdje
Lp6jn62mCVtlY9nS6sRkecpjuLYHRxyTdRu2iEVtSsjZqK55ovZ35SfkOPHe
FYa9BIuxWi9djHMVKQ8KOVGAVLibjZ6P8LreDSKtWDv9YQjIEnkwFVGCPfpB
q2SX4VTQfJAQXHggR8izKxPvluXUdG9rIPh4cac98++VLdIuFMiEXjUIoTX2
rEzunaCLMy0VIfowlRsgsKGrwhCCv7hBWyglbzwz+AAAAAAAAAQAAAF2SOr
JvPUerB9mtf4bqQDFthfoz/XepysnYuReHHEXKe+BFkaEoMNGiBl4TCLZx72
DvmZwKCewWRH1+W66ggrXKw2VdVl+vLsmzxNyWChGLfBTL5/3SUF09BfmCEl
03Ckk7htAgyAQcBf90RJznZndv7HwVAi3syupi0sQDdOKNPyObR5FRtqyqud
ttWmSdmGCGFcZ/fZqxQNsHB8QuYaBiGL7CDusES+wwfn8Q7BGtoJzOPDDx6K
yIyox/flPx2DZDJIZrMz9b0V70a9kqKLo/wcGhvHO6coCyMxenBAacLJ1DiI
NLKoYOoJTM7zcxsGnvCxaDZCvsmjx3j8Yc5r3i3ylllCQH2/lpr/xCvXFarG
tG7+wts+UqstS9SThLBQ9Ojq4oPsX7HBHKvq19XU3/ChIgWMy+bczc5gpkC/
eLAIGfJ0D5DJsl68vMXSmCoFK0HTwzzNa7lnZK4IutYPBNBCv0pWORQqDpsk
Ez96YOGyB8+gtpFgCrkuV1bSB9SRVmEBfDtKPQFhKowAAAAA=.
```

We could fragment this message into (for example) three pieces:

```
?OTR,1,3,?OTR:AAEDAAAAAQAAAAEAAADAVf3Ei72ZgFeKqWvLMnuVPVCwxk
tsOZ1QdjeLp6jn62mCVtlY9nS6sRkecpjuLYHRxyTdRu2iEVtSsjZqK55ovZ
35SfkOPHeFYa9BIuxWi9djHMVKQ8KOVGAVLibjZ6P8LreDSKtWDv9YQjIEnk
wFVGCPfpBq2SX4VTQfJAQXHggR8izKxPvluXUdG9rIPh4cac98++VLdIuFMi
EXjUIoTX2rEzunaCLMy0VIfowlRsgsKGrwhCCv7hBWyglbzwz+AAAAAAAAA
QAAAF2SOr,

?OTR,2,3,JvPUerB9mtf4bqQDFthfoz/XepysnYuReHHEXKe+BFkaEoMNGiB
l4TCLZx72DvmZwKCewWRH1+W66ggrXKw2VdVl+vLsmzxNyWChGLfBTL5/3SU
F09BfmCEl03Ckk7htAgyAQcBf90RJznZndv7HwVAi3syupi0sQDdOKNPyObR
5FRtqyqudttWmSdmGCGFcZ/fZqxQNsHB8QuYaBiGL7CDusES+wwfn8Q7BGto
JzOPDDx6KyIyox/flPx2DZDJIZrMz9b0V70a9kqKLo/wcGhvHO6coCyMxenB
AacLJ1DiI,

?OTR,3,3,NLKoYOoJTM7zcxsGnvCxaDZCvsmjx3j8Yc5r3i3ylllCQH2/lpr
/xCvXFarGtG7+wts+UqstS9SThLBQ9Ojq4oPsX7HBHKvq19XU3/ChIgWMy+b
czc5gpkC/eLAIGfJ0D5DJsl68vMXSmCoFK0HTwzzNa7lnZK4IutYPBNBCv0p
WORQqDpskEz96YOGyB8+gtpFgCrkuV1bSB9SRVmEBfDtKPQFhKowAAAAA=.,
```

The protocol state machine

An OTR client maintains separate state for every correspondent. For example, Alice may have an active OTR conversation with Bob, while having an unprotected conversation with Charlie. This state consists of two main state variables, as well as some other information (such as encryption keys). The two main state variables are:

Message state

The message state variable, msgstate, controls what happens to outgoing messages typed by the user. It can take one of three values:

MSGSTATE_PLAINTEXT
: This state indicates that outgoing messages are sent without encryption. This is the state that is used before an OTR conversation is initiated. This is the initial state, and the only way to subsequently enter this state is for the user to explicitly request to do so via some UI operation.

MSGSTATE_ENCRYPTED
: This state indicates that outgoing messages are sent encrypted. This is the state that is used during an OTR conversation. The only way to enter this state is for the authentication state machine (below) to successfully complete.

MSGSTATE_FINISHED
: This state indicates that outgoing messages are not delivered at all. This state is entered only when the other party indicates he has terminated his side of the OTR conversation. For example, if Alice and Bob are having an OTR conversation, and Bob instructs his OTR client to end its private session with Alice (for example, by logging out), Alice will be notified of this, and *her* client will switch to MSGSTATE_FINISHED mode. This prevents Alice from accidentally sending a message to Bob in plaintext. (Consider what happens if Alice was in the middle of typing a private message to Bob when he suddenly logs out, just as Alice hits Enter.)

Authentication state

The authentication state variable, authstate, can take one of four values (plus one extra for OTR version 1 compatibility):

AUTHSTATE_NONE
: This state indicates that the authentication protocol is not currently in progress. This is the initial state.

AUTHSTATE_AWAITING_DHKEY
: After Bob initiates the authentication protocol by sending Alice the D-H Commit Message, he enters this state to await Alice's reply.

AUTHSTATE_AWAITING_REVEALSIG
: After Alice receives Bob's D-H Commit Message, and replies with her own D-H Key Message, she enters this state to await Bob's reply.

AUTHSTATE_AWAITING_SIG
: After Bob receives Alice's D-H Key Message, and replies with his own Reveal Signature Message, he enters this state to await Alice's reply.

AUTHSTATE_V1_SETUP
: For OTR version 1 compatibility, if Bob sends a version 1 Key Exchange Message to Alice, he enters this state to await Alice's reply.

After:

- Alice (in AUTHSTATE_AWAITING_REVEALSIG) receives Bob's Reveal Signature Message (and replies with her own Signature Message),
- Alice (in AUTHSTATE_NONE) receives Bob's Version 1 Key Exchange Message (and replies with her own Key Exchange Message),
- Bob (in AUTHSTATE_AWAITING_SIG) receives Alice's Signature Message, **or**

- Bob (in AUTHSTATE_V1_SETUP) receives Alice's Version 1 Key Exchange Message,

then, assuming the signature verifications succeed, the msgstate variable is transitioned to MSGSTATE_ENCRYPTED. Regardless of whether the signature verifications succeed, the authstate variable is transitioned to AUTHSTATE_NONE.

Policies

OTR clients can set different **policies** for different correspondents. For example, Alice could set up her client so that it speaks only OTR version 2, except with Charlie, who she knows has only an old client; so that it will opportunistically start an OTR conversation whenever it detects the correspondent supports it; or so that it refuses to send non-encrypted messages to Bob, ever.

The policies that can be set (on a global or per-correspondent basis) are any combination of the following boolean flags:

ALLOW_V1
: Allow version 1 of the OTR protocol to be used.

ALLOW_V2
: Allow version 2 of the OTR protocol to be used.

REQUIRE_ENCRYPTION
: Refuse to send unencrypted messages.

SEND_WHITESPACE_TAG
: Advertise your support of OTR using the whitespace tag.

WHITESPACE_START_AKE
: Start the OTR AKE when you receive a whitespace tag.

ERROR_START_AKE
: Start the OTR AKE when you receive an OTR Error Message.

The four old version 1 policies correspond to the following combinations of flags (adding an allowance for version 2 of the protocol):

NEVER
: No flags set

MANUAL
: ALLOW_V1 | ALLOW_V2

OPPORTUNISTIC
: ALLOW_V1 | ALLOW_V2 | SEND_WHITESPACE_TAG | WHITESPACE_START_AKE | ERROR_START_AKE

ALWAYS
: ALLOW_V1 | ALLOW_V2 | REQUIRE_ENCRYPTION | WHITESPACE_START_AKE | ERROR_START_AKE

Note that it is possible for UIs simply to offer the old "combinations" of options, and not ask about each one separately.

State transitions

There are thirteen actions an OTR client must handle:

- Received messages:
    - Plaintext without the whitespace tag
    - Plaintext with the whitespace tag
    - Query Message
    - Error Message
    - D-H Commit Message
    - D-H Key Message
    - Reveal Signature Message
    - Signature Message
    - Version 1 Key Exchange Message
    - Data Message
- User actions:
    - User requests to start an OTR conversation
    - User requests to end an OTR conversation
    - User types a message to be sent

The following sections will outline what actions to take in each case. They all assume that at least one of ALLOW_V1 or ALLOW_V2 is set; if not, then OTR is completely disabled, and no special handling of messages should be done at all.

Receiving plaintext without the whitespace tag

If msgstate is MSGSTATE_PLAINTEXT:
> Simply display the message to the user. If REQUIRE_ENCRYPTION is set, warn him that the message was received unencrypted.

If msgstate is MSGSTATE_ENCRYPTED or MSGSTATE_FINISHED:
> Display the message to the user, but warn him that the message was received unencrypted.

Receiving plaintext with the whitespace tag

If msgstate is MSGSTATE_PLAINTEXT:
> Remove the whitespace tag and display the message to the user. If REQUIRE_ENCRYPTION is set, warn him that the message was received unencrypted.

If msgstate is MSGSTATE_ENCRYPTED or MSGSTATE_FINISHED:
> Remove the whitespace tag and display the message to the user, but warn him that the message was received unencrypted.

In any event, if WHITESPACE_START_AKE is set:

If the tag offers OTR version 2 and ALLOW_V2 is set:
> Send a D-H Commit Message, and transition authstate to AUTHSTATE_AWAITING_DHKEY.

Otherwise, if the tag offers OTR version 1 and ALLOW_V1 is set:
> Send a Version 1 Key Exchange Message, and transition authstate to AUTHSTATE_V1_SETUP.

Receiving a Query Message

If the Query Message offers OTR version 2 and ALLOW_V2 is set:

Send a D-H Commit Message, and transition authstate to AUTHSTATE_AWAITING_DHKEY.

Otherwise, if the message offers OTR version 1 and ALLOW_V1 is set:

Send a Version 1 Key Exchange Message, and transition authstate to AUTHSTATE_V1_SETUP.

Receiving an Error Message

Display the message to the user. If ERROR_START_AKE is set, reply with a Query Message.

User requests to start an OTR conversation

Send an OTR Query Message to the correspondent.

Receiving a D-H Commit Message

If ALLOW_V2 is not set, ignore this message. Otherwise:

If authstate is AUTHSTATE_NONE:

Reply with a D-H Key Message, and transition authstate to AUTHSTATE_AWAITING_REVEALSIG.

If authstate is AUTHSTATE_AWAITING_DHKEY:

This is the trickiest transition in the whole protocol. It indicates that you have already sent a D-H Commit message to your correspondent, but that he either didn't receive it, or just didn't receive it *yet*, and has sent you one as well. The symmetry will be broken by comparing the hashed g^x you sent in your D-H Commit Message with the one you received, considered as 32-byte unsigned big-endian values.

If yours is the higher hash value:

Ignore the incoming D-H Commit message, but resend your D-H Commit message.

Otherwise:

Forget your old g^x value that you sent (encrypted) earlier, and pretend you're in AUTHSTATE_NONE; i.e. reply with a D-H Key Message, and transition authstate to AUTHSTATE_AWAITING_REVEALSIG.

If authstate is AUTHSTATE_AWAITING_REVEALSIG:

Retransmit your D-H Key Message (the same one as you sent when you entered AUTHSTATE_AWAITING_REVEALSIG). Forget the old D-H Commit message, and use this new one instead. There are a number of reasons this might happen, including:

- Your correspondent simply started a new AKE.
- Your correspondent resent his D-H Commit message, as specified above.
- On some networks, like AIM, if your correspondent is logged in multiple times, each of his clients will send a D-H Commit Message in response to a Query Message; resending the same D-H Key Message in response to each of those messages will prevent compounded confusion, since each of his clients will see each of the D-H Key Messages you send. [And the problem gets even worse if you are *each* logged in multiple times.]

If authstate is AUTHSTATE_AWAITING_SIG or AUTHSTATE_V1_SETUP:

> Reply with a new D-H Key message, and transition authstate to AUTHSTATE_AWAITING_REVEALSIG.

Receiving a D-H Key Message

If ALLOW_V2 is not set, ignore this message. Otherwise:

If authstate is AUTHSTATE_AWAITING_DHKEY:

> Reply with a Reveal Signature Message and transition authstate to AUTHSTATE_AWAITING_SIG.

If authstate is AUTHSTATE_AWAITING_SIG:

> If this D-H Key message is the same the one you received earlier (when you entered AUTHSTATE_AWAITING_SIG):
>
> > Retransmit your Reveal Signature Message.
>
> Otherwise:
>
> > Ignore the message.

If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_REVEALSIG, or AUTHSTATE_V1_SETUP:

> Ignore the message.

Receiving a Reveal Signature Message

If ALLOW_V2 is not set, ignore this message. Otherwise:

If authstate is AUTHSTATE_AWAITING_REVEALSIG:

> Use the received value of r to decrypt the value of g^x received in the D-H Commit Message, and verify the hash therein. Decrypt the encrypted signature, and verify the signature and the MACs. If everything checks out:
>
> - Reply with a Signature Message.
> - Transition authstate to AUTHSTATE_NONE.
> - Transition msgstate to MSGSTATE_ENCRYPTED.
> - If there is a recent stored message, encrypt it and send it as a Data Message.
>
> Otherwise, ignore the message.

If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_DHKEY, AUTHSTATE_AWAITING_SIG, or AUTHSTATE_V1_SETUP:

> Ignore the message.

Receiving a Signature Message

If ALLOW_V2 is not set, ignore this message. Otherwise:

If authstate is AUTHSTATE_AWAITING_SIG:

> Decrypt the encrypted signature, and verify the signature and the MACs. If everything checks out:
>
> - Transition authstate to AUTHSTATE_NONE.
> - Transition msgstate to MSGSTATE_ENCRYPTED.
> - If there is a recent stored message, encrypt it and send it as a Data Message.
>
> Otherwise, ignore the message.

```
If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_DHKEY, AUTHSTATE_AWAITING_REVEALSIG, or AUTHSTATE_V1_SETUP:
    Ignore the message.
```

Receiving a Version 1 Key Exchange Message

If ALLOW_V1 is not set, ignore this message. Otherwise:

```
If authstate is AUTHSTATE_NONE, AUTHSTATE_AWAITING_DHKEY, AUTHSTATE_AWAITING_REVEALSIG, or AUTHSTATE_AWAITING_SIG:
    If the reply field is not set to 0x01:
        Verify the information in the Key Exchange Message. If the verification succeeds:
            • Reply with a Key Exchange Message with the reply field set to 0x01.
            • Transition authstate to AUTHSTATE_NONE.
            • Transition msgstate to MSGSTATE_ENCRYPTED.
            • If there is a recent stored message, encrypt it and send it as a Data Message.
        Otherwise, ignore the message.
    Otherwise, ignore the message.
If authstate is AUTHSTATE_V1_SETUP:
    Verify the information in the Key Exchange Message. If the verification succeeds:
        • If the received Key Exchange Message did not have the reply field set to 0x01, reply with a Key Exchange Message with the reply field set to 0x01.
        • Transition authstate to AUTHSTATE_NONE.
        • Transition msgstate to MSGSTATE_ENCRYPTED.
        • If there is a recent stored message, encrypt it and send it as a Data Message.
    Otherwise, ignore the message.
```

User types a message to be sent

```
If msgstate is MSGSTATE_PLAINTEXT:
    If REQUIRE_ENCRYPTION is set:
        Store the plaintext message for possible retransmission, and send a Query Message.
    Otherwise:
        If SEND_WHITESPACE_TAG is set, and you have not received a plaintext message from this correspondent since last entering MSGSTATE_PLAINTEXT, attach the whitespace tag to the message. Send the (possibly modified) message as plaintext.
If msgstate is MSGSTATE_ENCRYPTED:
    Encrypt the message, and send it as a Data Message. Store the plaintext message for possible retransmission.
If msgstate is MSGSTATE_FINISHED:
    Inform the user that the message cannot be sent at this time. Store the plaintext message for possible retransmission.
```

Receiving a Data Message

If msgstate is MSGSTATE_ENCRYPTED:

> Verify the information (MAC, keyids, ctr value, etc.) in the message.
>
> If the verification succeeds:
>
> - Decrypt the message and display the human-readable part (if non-empty) to the user.
> - Update the D-H encryption keys, if necessary.
> - If you have not sent a message to this correspondent in some (configurable) time, send a "heartbeat" message, consisting of a Data Message encoding an empty plaintext. The heartbeat message should have the IGNORE_UNREADABLE flag set.
> - If the received message contains a TLV type 1, forget all encryption keys for this correspondent, and transition msgstate to MSGSTATE_FINISHED.
>
> Otherwise, inform the user that an unreadable encrypted message was received, and reply with an Error Message.

If msgstate is MSGSTATE_PLAINTEXT or MSGSTATE_FINISHED:

> Inform the user that an unreadable encrypted message was received, and reply with an Error Message.

User requests to end an OTR conversation

If msgstate is MSGSTATE_PLAINTEXT:

> Do nothing.

If msgstate is MSGSTATE_ENCRYPTED:

> Send a Data Message, encoding a message with an empty human-readable part, and TLV type 1. Transition msgstate to MSGSTATE_PLAINTEXT.

If msgstate is MSGSTATE_FINISHED:

> Transition msgstate to MSGSTATE_PLAINTEXT.