1 Olm: A Cryptographic Ratchet

An implementation of the double cryptographic ratchet described by https://whispersystems.org/docs/specifica

1.1 Notation

This document uses ‘ ‖ ‘ to represent string concatenation. When ‘ ‖ ‘ appears on the right hand side of an ‘ = ‘ it means that the inputs are concatenated. When ‘ ‖ ‘ appears on the left hand side of an ‘ = ‘ it means that the output is split.

When this document uses ‘ $\text{ECDH}(K_A, K_B)$ ‘ it means that each party computes a Diffie-Hellman agreement using their private key and the remote party's public key. So party ‘A‘ computes ‘ $\text{ECDH}\left(K_B^{public}, K_A^{private}\right)$ ‘ and party ‘B‘ computes ‘ $\text{ECDH}\left(K_A^{public}, K_B^{private}\right)$ ‘.

Where this document uses ‘ $\text{HKDF}(salt, IKM, info, L)$ ‘ it refers to the HMAC-based key derivation function with a salt value of ‘$salt$‘, input key material of ‘IKM‘, context string ‘$info$‘, and output keying material length of ‘L‘ bytes.

1.2 The Olm Algorithm

1.2.1 Initial setup

The setup takes four Curve25519 inputs: Identity keys for Alice and Bob, ‘I_A‘ and ‘I_B‘, and one-time keys for Alice and Bob, ‘E_A‘ and ‘E_B‘. A shared secret, ‘S‘, is generated using Triple Diffie-Hellman. The initial 256 bit root key, ‘R_0‘, and 256 bit chain key, ‘$C_{0,0}$‘, are derived from the shared secret using an HMAC-based Key Derivation Function using SHA-256 as the hash function (HKDF-SHA-256) with default salt and `"OLM_ROOT"` as the info.

$$S = \text{ECDH}(I_A, E_B) \parallel \text{ECDH}(E_A, I_B) \parallel \text{ECDH}(E_A, E_B)$$

$$R_0 \parallel C_{0,0} = \text{HKDF}(0, S, \text{“OLM\_ROOT”}, 64)$$

1.2.2 Advancing the root key

Advancing a root key takes the previous root key, ‘R_{i-1}‘, and two Curve25519 inputs: the previous ratchet key, ‘T_{i-1}‘, and the current ratchet key ‘T_i‘. The even ratchet keys are generated by Alice. The odd ratchet keys are generated by Bob. A shared secret is generated using Diffie-Hellman on the ratchet keys. The next root key, ‘R_i‘, and chain key, ‘$C_{i,0}$‘, are derived from the shared secret using HKDF-SHA-256 using ‘R_{i-1}‘ as the salt and `"OLM_RATCHET"` as the info.

$$R_i \parallel C_{i,0} = \text{HKDF}(R_{i-1}, \text{ECDH}(T_{i-1}, T_i), \text{“OLM\_RATCHET”}, 64)$$

1.2.3 Advancing the chain key

Advancing a chain key takes the previous chain key, ‘$C_{i,j-1}$‘. The next chain key, ‘$C_{i,j}$‘, is the HMAC-SHA-256 of `"\x02"` using the previous chain key as

the key.

$C_{i,j} = \text{HMAC}\left(C_{i,j-1}, ““\text{x02}”\right)$

1.2.4 Creating a message key

Creating a message key takes the current chain key, ‘$C_{i,j}$‘. The message key, ‘$M_{i,j}$‘, is the HMAC-SHA-256 of `"\x01"` using the current chain key as the key. The message keys where ‘i‘ is even are used by Alice to encrypt messages. The message keys where ‘i‘ is odd are used by Bob to encrypt messages.

$M_{i,j} = \text{HMAC}\left(C_{i,j}, ““\text{x01}”\right)$

1.3 The Olm Protocol

1.3.1 Creating an outbouns session

Bob publishes the public parts of his identity key, ‘I_B‘, and some single-use one-time keys ‘E_B‘.

Alice downloads Bob's identity key, ‘I_B‘, and a one-time key, ‘E_B‘. She generates a new single-use key, ‘E_A‘, and computes a root key, ‘R_0‘, and a chain key ‘$C_{0,0}$‘. She also generates a new ratchet key ‘T_0‘.

1.3.2 Sending the first pre-key messages

Alice computes a message key, ‘$M_{0,j}$‘, and a new chain key, ‘$C_{0,j+1}$‘, using the current chain key. She replaces the current chain key with the new one.

Alice encrypts her plain-text with the message key, ‘$M_{0,j}$‘, using an authenticated encryption scheme (see below) to get a cipher-text, ‘$X_{0,j}$‘.

She then sends the following to Bob: * The public part of her identity key, ‘I_A‘ * The public part of her single-use key, ‘E_A‘ * The public part of Bob's single-use key, ‘E_B‘ * The current chain index, ‘j‘ * The public part of her ratchet key, ‘T_0‘ * The cipher-text, ‘$X_{0,j}$‘

Alice will continue to send pre-key messages until she receives a message from Bob.

1.3.3 Creating an inbound session from a pre-key message

Bob receives a pre-key message as above.

Bob looks up the private part of his single-use key, ‘E_B‘. He can now compute the root key, ‘R_0‘, and the chain key, ‘$C_{0,0}$‘, from ‘I_A‘, ‘E_A‘, ‘I_B‘, and ‘E_B‘.

Bob then advances the chain key ‘j‘ times, to compute the chain key used by the message, ‘$C_{0,j}$‘. He now creates the message key, ‘$M_{0,j}$‘, and attempts to decrypt the cipher-text, ‘$X_{0,j}$‘. If the cipher-text's authentication is correct then Bob can discard the private part of his single-use one-time key, ‘E_B‘.

Bob stores Alice's initial ratchet key, ‘T_0‘, until he wants to send a message.

1.3.4 Sending normal messages

Once a message has been received from the other side, a session is considered established, and a more compact form is used.

To send a message, the user checks if they have a sender chain key, '$C_{i,j}$'. Alice uses chain keys where 'i' is even. Bob uses chain keys where 'i' is odd. If the chain key doesn't exist then a new ratchet key 'T_i' is generated and a new root key 'R_i' and chain key '$C_{i,0}$' are computed using 'R_{i-1}', 'T_{i-1}' and 'T_i'.

A message key, '$M_{i,j}$' is computed from the current chain key, '$C_{i,j}$', and the chain key is replaced with the next chain key, '$C_{i,j+1}$'. The plain-text is encrypted with '$M_{i,j}$', using an authenticated encryption scheme (see below) to get a cipher-text, '$X_{i,j}$'.

The user then sends the following to the recipient: * The current chain index, 'j' * The public part of the current ratchet key, 'T_i' * The cipher-text, '$X_{i,j}$'

1.3.5 Receiving messages

The user receives a message as above with the sender's current chain index, 'j', the sender's ratchet key, 'T_i', and the cipher-text, '$X_{i,j}$'.

The user checks if they have a receiver chain with the correct 'i' by comparing the ratchet key, 'T_i'. If the chain doesn't exist then they compute a new root key, 'R_i', and a new receiver chain, with chain key '$C_{i,0}$', using 'R_{i-1}', 'T_{i-1}' and 'T_i'.

If the 'j' of the message is less than the current chain index on the receiver then the message may only be decrypted if the receiver has stored a copy of the message key '$M_{i,j}$'. Otherwise the receiver computes the chain key, '$C_{i,j}$'. The receiver computes the message key, '$M_{i,j}$', from the chain key and attempts to decrypt the cipher-text, '$X_{i,j}$'.

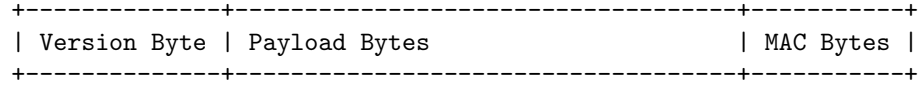
If the decryption succeeds the receiver updates the chain key for 'T_i' with '$C_{i,j+1}$' and stores the message keys that were skipped in the process so that they can decode out of order messages. If the receiver created a new receiver chain then they discard their current sender chain so that they will create a new chain when they next send a message.

1.4 The Olm Message Format

Olm uses two types of messages. The underlying transport protocol must provide a means for recipients to distinguish between them.

1.4.1 Normal Messages

Olm messages start with a one byte version followed by a variable length payload followed by a fixed length message authentication code.



The version byte is `"\x03"`.

The payload consists of key-value pairs where the keys are integers and the values are integers and strings. The keys are encoded as a variable length integer tag where the 3 lowest bits indicates the type of the value: 0 for integers, 2 for strings. If the value is an integer then the tag is followed by the value encoded as a variable length integer. If the value is a string then the tag is followed by the length of the string encoded as a variable length integer followed by the string itself.

Olm uses a variable length encoding for integers. Each integer is encoded as a sequence of bytes with the high bit set followed by a byte with the high bit clear. The seven low bits of each byte store the bits of the integer. The least significant bits are stored in the first byte.

Name	Tag	Type	Meaning
Ratchet-Key	0x0A	String	The public part of the ratchet key, T_i, of the message
Chain-Index	0x10	Integer	The chain index, j, of the message
Cipher-Text	0x22	String	The cipher-text, $X_{i,j}$, of the message

The length of the MAC is determined by the authenticated encryption algorithm being used. (Olm version 1 uses HMAC-SHA-256, truncated to 8 bytes). The MAC protects all of the bytes preceding the MAC.

1.4.2 Pre-Key Messages

Olm pre-key messages start with a one byte version followed by a variable length payload.

```
+--------------+------------------------------------+
| Version Byte | Payload Bytes                      |
+--------------+------------------------------------+
```

The version byte is `"\x03"`.

The payload uses the same key-value format as for normal messages.

Name	Tag	Type	Meaning
One-Time-Key	0x0A	String	The public part of Bob's single-use key, Eb.
Base-Key	0x12	String	The public part of Alice's single-use key, Ea.
Identity-Key	0x1A	String	The public part of Alice's identity key, Ia.
Message	0x22	String	An embedded Olm message with its own version and MAC.

1.5 Olm Authenticated Encryption

1.5.1 Version 1

Version 1 of Olm uses AES-256 in CBC mode with PKCS#7 padding for encryption and HMAC-SHA-256 (truncated to 64 bits) for authentication. The 256 bit AES key, 256 bit HMAC key, and 128 bit AES IV are derived from the message key using HKDF-SHA-256 using the default salt and an info of `"OLM_KEYS"`.

$AES\_KEY_{i,j} \parallel HMAC\_KEY_{i,j} \parallel AES\_IV_{i,j} = \text{HKDF}(0, M_{i,j}, \text{“OLM\_KEYS”}, 80)$

The plain-text is encrypted with AES-256, using the key '$AES\_KEY_{i,j}$' and the IV '$AES\_IV_{i,j}$' to give the cipher-text, '$X_{i,j}$'.

Then the entire message (including the Version Byte and all Payload Bytes) are passed through HMAC-SHA-256. The first 8 bytes of the MAC are appended to the message.

1.6 Message authentication concerns

To avoid unknown key-share attacks, the application must include identifying data for the sending and receiving user in the plain-text of (at least) the pre-key messages. Such data could be a user ID, a telephone number; alternatively it could be the public part of a keypair which the relevant user has proven ownership of.

1.6.1 Example attacks

1. Alice publishes her public Curve25519 identity key, 'I_A'. Eve publishes the same identity key, claiming it as her own. Bob downloads Eve's keys, and associates 'I_A' with Eve. Alice sends a message to Bob; Eve intercepts it before forwarding it to Bob. Bob believes the message came from Eve rather than Alice.

   This is prevented if Alice includes her user ID in the plain-text of the pre-key message, so that Bob can see that the message was sent by Alice originally.

2. Bob publishes his public Curve25519 identity key, 'I_B'. Eve publishes the same identity key, claiming it as her own. Alice downloads Eve's keys, and associates 'I_B' with Eve. Alice sends a message to Eve; Eve cannot decrypt it, but forwards it to Bob. Bob believes the Alice sent the message to him, wheras Alice intended it to go to Eve.

   This is prevented by Alice including the user ID of the intended recpient (Eve) in the plain-text of the pre-key message. Bob can now tell that the message was meant for Eve rather than him.

1.7 IPR

The Olm specification (this document) is hereby placed in the public domain.

1.8 Feedback

Can be sent to olm at matrix.org.

1.9 Acknowledgements

The ratchet that Olm implements was designed by Trevor Perrin and Moxie Marlinspike - details at https://whispersystems.org/docs/specifications/doubleratchet/. Olm is an entirely new implementation written by the Matrix.org team.