1 Megolm group ratchet

An AES-based cryptographic ratchet intended for group communications.

1.1 Background

The Megolm ratchet is intended for encrypted messaging applications where there may be a large number of recipients of each message, thus precluding the use of peer-to-peer encryption systems such as Olm.

It also allows a recipient to decrypt received messages multiple times. For instance, in client/server applications, a copy of the ciphertext can be stored on the (untrusted) server, while the client need only store the session keys.

1.2 Overview

Each participant in a conversation uses their own outbound session for encrypting messages. A session consists of a ratchet and an Ed25519 keypair.

Secrecy is provided by the ratchet, which can be wound forwards but not backwards, and is used to derive a distinct message key for each message.

Authenticity is provided via Ed25519 signatures.

The value of the ratchet, and the public part of the Ed25519 key, are shared with other participants in the conversation via secure peer-to-peer channels. Provided that peer-to-peer channel provides authenticity of the messages to the participants and deniability of the messages to third parties, the Megolm session will inherit those properties.

1.3 The Megolm ratchet algorithm

The Megolm ratchet R_i consists of four parts, $R_{i,j}$ for $j \in 0, 1, 2, 3$. The length of each part depends on the hash function in use (256 bits for this version of Megolm).

The ratchet is initialised with cryptographically-secure random data, and advanced as follows:

$$R_{i,0} = \begin{cases} H_0\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ R_{i-1,0} & \text{otherwise} \end{cases}$$

$$R_{i,1} = \begin{cases} H_1\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ H_1\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m \\ R_{i-1,1} & \text{otherwise} \end{cases}$$

$$R_{i,2} = \begin{cases} H_2\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ H_2\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m \\ H_2\left(R_{2^{8}(p-1),2}\right) & \text{if } \exists p | i = 2^{8}p \\ R_{i-1,2} & \text{otherwise} \end{cases}$$

$$R_{i,3} = \begin{cases} H_3\left(R_{2^{24}(n-1),0}\right) & \text{if } \exists n | i = 2^{24}n \\ H_3\left(R_{2^{16}(m-1),1}\right) & \text{if } \exists m | i = 2^{16}m \\ H_3\left(R_{2^{8}(p-1),2}\right) & \text{if } \exists p | i = 2^{8}p \\ H_3\left(R_{i-1,3}\right) & \text{otherwise} \end{cases}$$

where H_0, H_1, H_2, and H_3 are different hash functions. In summary: every 2^8 iterations, $R_{i,3}$ is reseeded from $R_{i,2}$. Every 2^{16} iterations, $R_{i,2}$ and $R_{i,3}$ are reseeded from $R_{i,1}$. Every 2^{24} iterations, $R_{i,1}$, $R_{i,2}$ and $R_{i,3}$ are reseeded from $R_{i,0}$.

The complete ratchet value, R_i, is hashed to generate the keys used to encrypt each message. This scheme allows the ratchet to be advanced an arbitrary amount forwards while needing at most 1020 hash computations. A client can decrypt chat history onwards from the earliest value of the ratchet it is aware of, but cannot decrypt history from before that point without reversing the hash function.

This allows a participant to share its ability to decrypt chat history with another from a point in the conversation onwards by giving a copy of the ratchet at that point in the conversation.

1.4 The Megolm protocol

1.4.1 Session setup

Each participant in a conversation generates their own Megolm session. A session consists of three parts:

- a 32 bit counter, i.
- an Ed25519 keypair, K.
- a ratchet, R_i, which consists of four 256-bit values, $R_{i,j}$ for $j \in 0, 1, 2, 3$.

The counter i is initialised to 0. A new Ed25519 keypair is generated for K. The ratchet is simply initialised with 1024 bits of cryptographically-secure random data.

A single participant may use multiple sessions over the lifetime of a conversation. The public part of K is used as an identifier to discriminate between sessions.

1.4.2 Sharing session data

To allow other participants in the conversation to decrypt messages, the session data is formatted as described in 1.5.1Session-sharing format. It is then shared with other participants in the conversation via a secure peer-to-peer channel (such as that provided by Olm).

When the session data is received from other participants, the recipient first checks that the signature matches the public key. They then store their own copy of the counter, ratchet, and public key.

1.4.3 Message encryption

This version of Megolm uses AES-256 in CBC mode with PKCS#7 padding and HMAC-SHA-256 (truncated to 64 bits). The 256 bit AES key, 256 bit HMAC key, and 128 bit AES IV are derived from the megolm ratchet R_i:

$$AES\_KEY_i \parallel HMAC\_KEY_i \parallel AES\_IV_i = \text{HKDF}\,(0,\, R_i, \text{"MEGOLM\_KEYS"},\, 80)$$

where $\parallel$ represents string splitting, and $\text{HKDF}\,(salt,\, IKM,\, info,\, L)$ refers to the HMAC-based key derivation function using using SHA-256 as the hash function (HKDF-SHA-256) with a salt value of *salt*, input key material of *IKM*, context string *info*, and output keying material length of L bytes.

The plain-text is encrypted with AES-256, using the key AES_KEY_i and the IV AES_IV_i to give the cipher-text, X_i.

The ratchet index i, and the cipher-text X_i, are then packed into a message as described in 1.5.2Message format. Then the entire message (including the version bytes and all payload bytes) are passed through HMAC-SHA-256. The first 8 bytes of the MAC are appended to the message.

Finally, the authenticated message is signed using the Ed25519 keypair; the 64 byte signature is appended to the message.

The complete signed message, together with the public part of K (acting as a session identifier), can then be sent over an insecure channel. The message can then be authenticated and decrypted only by recipients who have received the session data.

1.4.4 Advancing the ratchet

After each message is encrypted, the ratchet is advanced. This is done as described in 1.3The Megolm ratchet algorithm, using the following definitions:

$H_0(A) \equiv \text{HMAC}(A, \text{""x00"})$

$H_1(A) \equiv \text{HMAC}(A, \text{""x01"})$

$H_2(A) \equiv \text{HMAC}(A, \text{""x02"})$

$H_3(A) \equiv \text{HMAC}(A, \text{""x03"})$

where $\text{HMAC}(A, T)$ is the HMAC-SHA-256 of `T`, using `A` as the key.

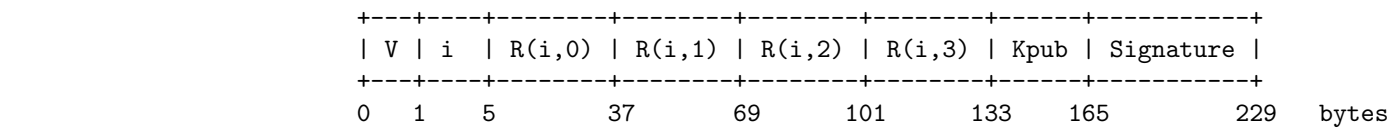
For outbound sessions, the updated ratchet and counter are stored in the session.

In order to maintain the ability to decrypt conversation history, inbound sessions should store a copy of their earliest known ratchet value (unless they explicitly want to drop the ability to decrypt that history - see 1.6.4Partial Forward Secrecy). They may also choose to cache calculated ratchet values, but the decision of which ratchet states to cache is left to the application.

1.5 Data exchange formats

1.5.1 Session-sharing format

The Megolm key-sharing format is as follows:



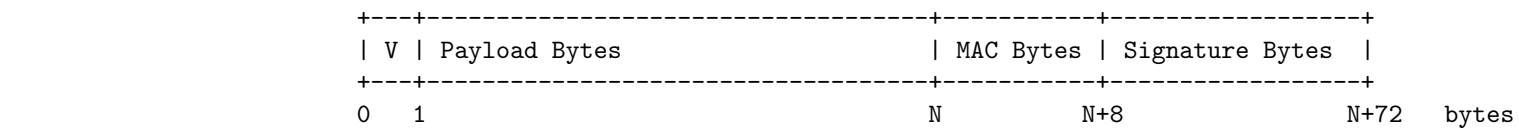
The version byte, `V`, is `"\x02"`.

This is followed by the ratchet index, i, which is encoded as a big-endian 32-bit integer; the ratchet values $R_{i,j}$; and the public part of the Ed25519 keypair K.

The data is then signed using the Ed25519 keypair, and the 64-byte signature is appended.

1.5.2 Message format

Megolm messages consist of a one byte version, followed by a variable length payload, a fixed length message authentication code, and a fixed length signature.



The version byte, `V`, is `"\x03"`.

The payload uses a format based on the Protocol Buffers encoding. It consists of the following key-value pairs:

Name	Tag	Type	Meaning
Message-Index	0x08	Integer	The index of the ratchet, i
Cipher-Text	0x12	String	The cipher-text, X_i, of the message

Within the payload, integers are encoded using a variable length encoding. 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.

Strings are encoded as a variable-length integer followed by the string itself.

Each key-value pair is encoded as a variable-length integer giving the tag, followed by a string or variable-length integer giving the value.

The payload is followed by the MAC. The length of the MAC is determined by the authenticated encryption algorithm being used (8 bytes in this version of the protocol). The MAC protects all of the bytes preceding the MAC.

The length of the signature is determined by the signing algorithm being used (64 bytes in this version of the protocol). The signature covers all of the bytes preceding the signature.

1.6 Limitations

1.6.1 Message Replays

A message can be decrypted successfully multiple times. This means that an attacker can re-send a copy of an old message, and the recipient will treat it as a new message.

To mitigate this it is recommended that applications track the ratchet indices they have received and that they reject messages with a ratchet index that they have already decrypted.

1.6.2 Lack of Transcript Consistency

In a group conversation, there is no guarantee that all recipients have received the same messages. For example, if Alice is in a conversation with Bob and Charlie, she could send different messages to Bob and Charlie, or could send some messages to Bob but not Charlie, or vice versa.

Solving this is, in general, a hard problem, particularly in a protocol which does not guarantee in-order message delivery. For now it remains the subject of future research.

1.6.3 Lack of Backward Secrecy

Backward secrecy (also called 'future secrecy' or 'post-compromise security') is the property that if current private keys are compromised, an attacker cannot decrypt future messages in a given session. In other words, when looking **backwards** in time at a compromise which has already happened, **current** messages are still secret.

By itself, Megolm does not possess this property: once the key to a Megolm session is compromised, the attacker can decrypt any message that was encrypted using a key derived from the compromised or subsequent ratchet values.

In order to mitigate this, the application should ensure that Megolm sessions are not used indefinitely. Instead it should periodically start a new session, with new keys shared over a secure channel.

1.6.4 Partial Forward Secrecy

Forward secrecy (also called 'perfect forward secrecy') is the property that if the current private keys are compromised, an attacker cannot decrypt *past* messages in a given session. In other words, when looking **forwards** in time towards a potential future compromise, **current** messages will be secret.

In Megolm, each recipient maintains a record of the ratchet value which allows them to decrypt any messages sent in the session after the corresponding point in the conversation. If this value is compromised, an attacker can similarly decrypt past messages which were encrypted by a key derived from the compromised or subsequent ratchet values. This gives 'partial' forward secrecy.

To mitigate this issue, the application should offer the user the option to discard historical conversations, by winding forward any stored ratchet values, or discarding sessions altogether.

1.6.5 Dependency on secure channel for key exchange

The design of the Megolm ratchet relies on the availability of a secure peer-to-peer channel for the exchange of session keys. Any vulnerabilities in the underlying channel are likely to be amplified when applied to Megolm session setup.

For example, if the peer-to-peer channel is vulnerable to an unknown key-share attack, the entire Megolm session become similarly vulnerable. For example: Alice starts a group chat with Eve, and shares the session keys with Eve. Eve uses the unknown key-share attack to forward the session keys to Bob, who believes Alice is starting the session with him. Eve then forwards messages from the Megolm session to Bob, who again believes they are coming from Alice. Provided the peer-to-peer channel is not vulnerable to this attack, Bob will realise that the key-sharing message was forwarded by Eve, and can treat the Megolm session as a forgery.

A second example: if the peer-to-peer channel is vulnerable to a replay attack, this can be extended to entire Megolm sessions.

1.7 License