End-to-End Encryption, Secret Chats

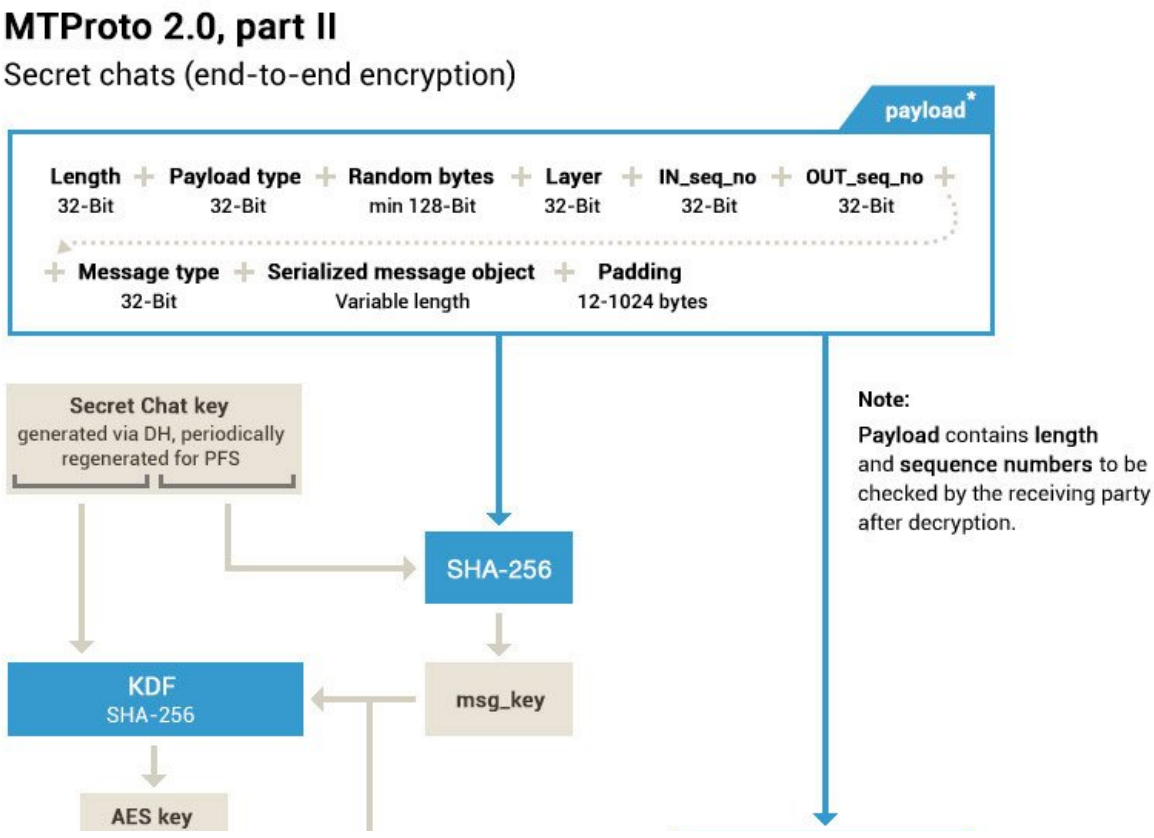
> This article on MTProto's End-to-End encryption is meant for advanced users. If you want to learn more about Secret Chats from a less intimidating source, kindly see our general FAQ.

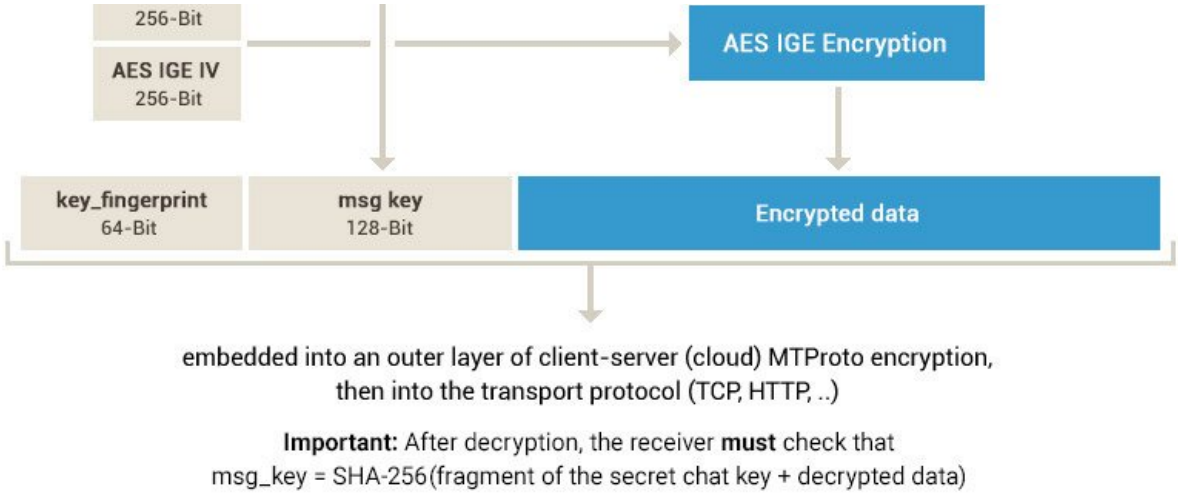
> Note that as of version 4.6, major Telegram clients are using MTProto 2.0. MTProto v.1.0 is deprecated and is currently being phased out.

Related articles

Security guidelines for developers
Perfect Forward Secrecy in Secret Chats
Sequence numbers in Secret Chats
End-to-End TL Schema

Secret Chats are one-on-one chats wherein messages are encrypted with a key held only by the chat's participants. Note that the schema for these end-to-end encrypted Secret Chats is different from what is used for cloud chats:





A note on MTProto 2.0

This article describes the end-to-end encryption layer in the MTProto protocol version 2.0. The principal differences from version 1.0 (described here for reference) are as follows:

- SHA-256 is used instead of SHA-1;
- Padding bytes are involved in the computation of msg_key;
- msg_key depends not only on the message to be encrypted, but on a portion of the secret chat key as well;
- 12..1024 padding bytes are used instead of 0..15 padding bytes in v.1.0.

See also: MTProto 2.0: Cloud Chats, server-client encryption

Key Generation

Keys are generated using the Diffie-Hellman protocol.

Let us consider the following scenario: User A would like to initiate end-to-end encrypted communication with User B.

Sending a Request

User A executes messages.getDhConfig to obtain the Diffie-Hellman parameters: a prime p, and a high order element g.

Executing this method before each new key generation procedure is of vital importance. It makes sense to cache the values of the parameters together with the version in order to avoid having to receive all of the values every time. If the version stored on the client is still up-to-date, the server will return the constructor messages.dhConfigNotModified.

Client is expected to check whether p is a safe 2048-bit prime (meaning that both p and (p-1)/2 are prime, and that 2^2047 < p < 2^2048), and that g generates a cyclic subgroup of prime order (p-1)/2, i.e. is a quadratic residue mod p. Since g is always equal to 2, 3, 4, 5, 6 or 7, this is easily done using quadratic reciprocity law, yielding a simple condition on p mod 4g -- namely, p mod 8 = 7 for g = 2; p mod 3 = 2 for g = 3; no extra condition for g = 4; p mod 5 = 1 or 4 for g = 5; p mod 24 = 19 or 23 for g = 6; and p mod 7 = 3, 5 or 6 for g = 7. After g and p have been checked by the client, it makes sense to cache the result, so as to avoid repeating lengthy computations in future. This cache might be shared with one used for Authorization Key generation.

If the client has an inadequate random number generator, it makes sense to pass the random_length parameter (random_length> 0) so the server generates its own random sequence random of the appropriate length. Important: using the server's random sequence in its raw form may be unsafe. It must be combined with a client sequence, for example, by generating a client random number of the same length (client_random) and using `final_random := random XOR client_random`.

Client A computes a 2048-bit number a (using sufficient entropy or the server's random; see above) and executes messages.requestEncryption after passing in `g_a := pow(g, a) mod dh_prime`.

User B receives the update updateEncryption for all associated authorization keys (all authorized devices) with the chat constructor encryptedChatRequested. The user must be shown basic information about User A and must be prompted to accept or reject the request.

Both clients are to check that g, g_a and g_b are greater than one and smaller than p-1. We recommend checking that g_a and g_b are between 2^{2048-64} and p - 2^{2048-64} as well.

Accepting a Request

After User B confirms the creation of a secret chat with A in the client interface, Client B also receives up-to-date configuration parameters for the Diffie-Hellman method. Thereafter, it generates a random 2048-bit number, b, using rules similar to those for a.

Having received g_a from the update with encryptedChatRequested, it can immediately generate the final shared key: `key = (pow(g_a, b) mod dh_prime)`. If key length < 256 bytes, add several leading zero bytes as padding — so that the key is exactly 256 bytes long. Its fingerprint, key_fingerprint, is equal to the 64 last bits of SHA1 (key).

Note 1: in this particular case SHA1 is used here even for MTProto 2.0 secret chats.

Note 2: this fingerprint is used as a sanity check for the key exchange procedure to detect bugs when developing client software — it is not connected to the key visualization used on the clients as means of external authentication in secret chats. Key visualizations on the clients are generated using the first 128 bits of SHA1(intial key) followed by the first 160 bits of SHA256(key used when secret chat was updated to layer 46).

Client B executes messages.acceptEncryption after passing it `g_b := pow(g, b) mod dh_prime` and key_fingerprint.

For all of Client B's authorized devices, except the current one, updateEncryption updates are sent with the constructor encryptedChatDiscarded. Thereafter, the only device that will be able to access the secret chat is Device B, which made the call to messages.acceptEncryption.

User A will be sent an updateEncryption update with the constructor encryptedChat, for the authorization key that initiated the chat.

With g_b from the update, Client A can also compute the shared key `key = (pow(g_b, a) mod dh_prime)`. If key length < 256 bytes, add several leading zero bytes as padding — so that the key is exactly 256 bytes long. If the fingerprint for the received key is identical to the one that was passed to encryptedChat, incoming messages can be sent and processed. Otherwise, messages.discardEncryption must be executed and the user notified.

Perfect Forward Secrecy

In order to keep past communications safe, official Telegram clients will initiate re-keying once a key

has been used to decrypt and encrypt more than 100 messages, or has been in use for more than one week, provided the key has been used to encrypt at least one message. Old keys are then securely discarded and cannot be reconstructed, even with access to the new keys currently in use.

> The re-keying protocol is further described in this article: Perfect Forward Secrecy in Secret Chats.

Please note that your client must support Forward Secrecy in Secret Chats to be compatible with official Telegram clients.

Sending and Receiving Messages in a Secret Chat

Serialization and Encryption of Outgoing Messages

A TL object of type DecryptedMessage is created and contains the message in plain text. For backward compatibility, the object must be wrapped in the constructor decryptedMessageLayer with an indication of the supported layer (starting with 46).

> The TL-Schema for the contents of end-to-end encrypted messages is available here »

The resulting construct is serialized as an array of bytes using generic TL rules. The resulting array is prepended by 4 bytes containing the array length not counting these 4 bytes.

The byte array is padded with 12 to 1024 random padding bytes to make its length divisible by 16 bytes. (In the older MTProto 1.0 encryption, only 0 to 15 padding bytes were used.)

Message key, msg_key, is computed as the 128 middle bits of the SHA256 of the data obtained in the previous step, prepended by 32 bytes from the shared key key. (For the older MTProto 1.0 encryption, msg_key was computed differently, as the 128 lower bits of SHA1 of the data obtained in the previous steps, excluding the padding bytes.)

For MTProto 2.0, the AES key aes_key and initialization vector aes_iv are computed (key is the shared key obtained during Key Generation) as follows:

- msg_key_large = SHA256 (substr (key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (key, x, 36));
- sha256_b = SHA256 (substr (key, 40+x, 36) + msg_key);
- aes_key = substr (sha256_a, 0, 8) + substr (sha256_b, 8, 16) + substr (sha256_a, 24, 8);
- aes_iv = substr (sha256_b, 0, 8) + substr (sha256_a, 8, 16) + substr (sha256_b, 24, 8);

For MTProto 2.0, x=0 for messages from the originator of the secret chat, x=8 for the messages in the opposite direction.

For the obsolete MTProto 1.0, msg_key, aes_key, and aes_iv were computed differently (see this document for reference).

Data is encrypted with a 256-bit key, aes_key, and a 256-bit initialization vector, aes-iv, using AES-256 encryption with infinite garble extension (IGE). Encryption key fingerprint key_fingerprint and the message key msg_key are added at the top of the resulting byte array.

Encrypted data is embedded into a messages.sendEncrypted API call and passed to Telegram server for delivery to the other party of the Secret Chat.

Upgrading to MTProto 2.0 from MTProto 1.0

As soon as both parties in a secret chat are using at least Layer 73, they should only use MTProto 2.0 for all outgoing messages. Some of the first received messages may use MTProto 1.0, if a sufficiently high starting layer has not been negotiated during the creation of the secret chat. After the first message encrypted with MTProto 2.0 (or the first message with Layer 73 or higher) is received, all messages with higher sequence numbers must be encrypted with MTProto 2.0 as well.

As long as the current layer is lower than 73, each party should try to decrypt received messages with MTProto 1.0, and if this is not successfull (msg_key does not match), try MTProto 2.0. Once the first MTProto 2.0-encrypted message arrives (or the layer is upgraded to 73), there is no need to try MTProto 1.0 decryption for any of the further messages (unless the client is still waiting for some gaps to be closed).

Decrypting an Incoming Message

The steps above are performed in reverse order. When an encrypted message is received, you must check that msg_key is in fact equal to the 128 middle bits of the SHA256 hash of the decrypted message, prepended by 32 bytes taken from the shared key. If the message layer is greater than the one supported by the client, the user must be notified that the client version is out of date and prompted to update.

Sequence numbers

It is necessary to interpret all messages in their original order to protect against possible manipulations. Secret chats support a special mechanism for handling seq_no counters independently from the server.

> Proper handling of these counters is further described in this article: Sequence numbers in Secret Chats.

Please note that your client must support sequence numbers in Secret Chats to be compatible with official Telegram clients.

Sending Encrypted Files

All files sent to secret chats are encrypted with one-time keys that are in no way related to the chat's shared key. Before an encrypted file is sent, it is assumed that the encrypted file's address will be attached to the outside of an encrypted message using the file parameter of the messages.sendEncryptedFile method and that the key for direct decryption will be sent in the body of the message (the key parameter in the constructors decryptedMessageMediaPhoto, decryptedMessageMediaVideo and decryptedMessageMediaFile.

Prior to a file being sent to a secret chat, 2 random 256-bit numbers are computed which will serve as the AES key and initialization vector used to encrypt the file. AES-256 encryption with infinite garble extension (IGE) is used in like manner.

The key fingerprint is computed as follows:

- digest = md5(key + iv)
- fingerprint = substr(digest, 0, 4) XOR substr(digest, 4, 4)

The encrypted contents of a file are stored on the server in much the same way as those of a file in

cloud chats: piece by piece using calls to upload.saveFilePart. A subsequent call to messages.sendEncryptedFile will assign an identifier to the stored file and send the address together with the message. The recipient will receive an update with encryptedMessage, and the file parameter will contain file information.

Incoming and outgoing encrypted files can be forwarded to other secret chats using the constructor inputEncryptedFile to avoid saving the same content on the server twice.

Working with an Update Box

Secret chats are associated with specific devices (or rather with authorization keys), not users. A conventional message box, which uses pts to describe the client's status, is not suitable, because it is designed for long-term message storage and message access from different devices.

An additional temporary message queue is introduced as a solution to this problem. When an update regarding a message from a secret chat is sent, a new value of qts is sent, which helps reconstruct the difference if there has been a long break in the connection or in case of loss of an update.

As the number of events increases, the value of qts increases by 1 with each new event. The initial value may not (and will not) be equal to 0.

The fact that events from the temporary queue have been received and stored by the client is acknowledged explicitly by a call to the messages.receivedQueue method or implicitly by a call to updates.getDifference (the value of qts passed, not the final state). All messages acknowledged as delivered by the client, as well as any messages older than 7 days, may (and will) be deleted from the server.

Upon de-authorization, the event queue of the corresponding device will be forcibly cleared, and the value of qts will become irrelevant.

Updating to new layers

Your client should always store the maximal layer that is known to be supported by the client on the other side of a secret chat. When the secret chat is first created, this value should be initialized to 46. This remote layer value must always be updated immediately after receiving any packet containing information of an upper layer, i.e.:

- any secret chat message containing layer_no in its `decryptedMessageLayer` with layer>=46, or
- a decryptedMessageActionNotifyLayer service message, wrapped as if it were the decryptedMessageService constructor of the obsolete layer 8 (constructor `decryptedMessageService#aa48327d`).

Notifying the remote client about your local layer

In order to notify the remote client of your local layer, your client must send a message of the `decryptedMessageActionNotifyLayer` type. This notification must be wrapped in a constructor of an appropriate layer.

There are two cases when your client must notify the remote client about its local layer:

1. As soon as a new secret chat has been created, immediately after the secret key has been successfully exchanged.
2. Immediately after the local client has been updated to support a new secret chat layer. In this case notifications must be sent to all currently existing secret chats. Note that this is only

necessary when updating to new layers that contain changes in the secret chats implementation (e.g. you don't need to do this when your client is updated from Layer 46 to Layer 47).