Mobile Protocol: Detailed Description

> 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.

This article describes the basic layer of the MTProto protocol version 2.0 (Cloud chats, server-client encryption). 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 auth_key as well;
- 12..1024 padding bytes are used instead of 0..15 padding bytes in v.1.0.

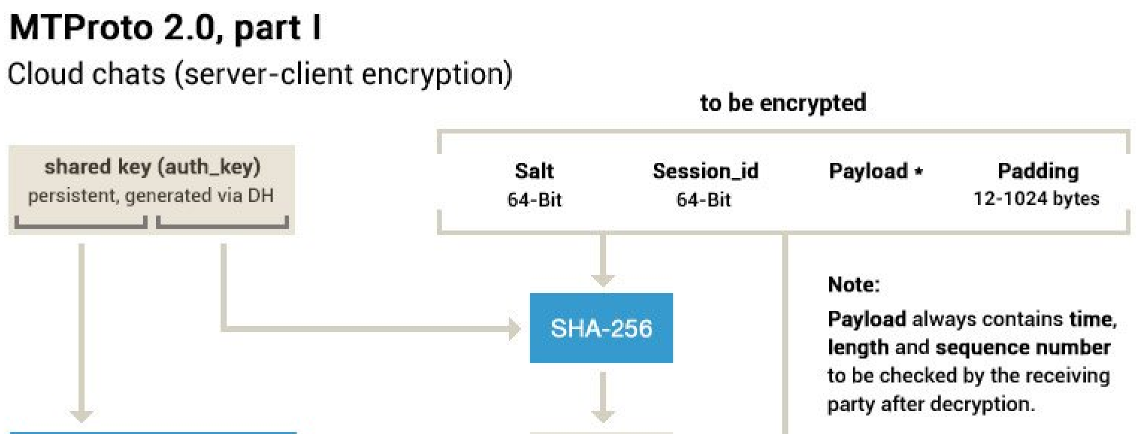
See also: MTProto 2.0: Secret Chats, end-to-end encryption

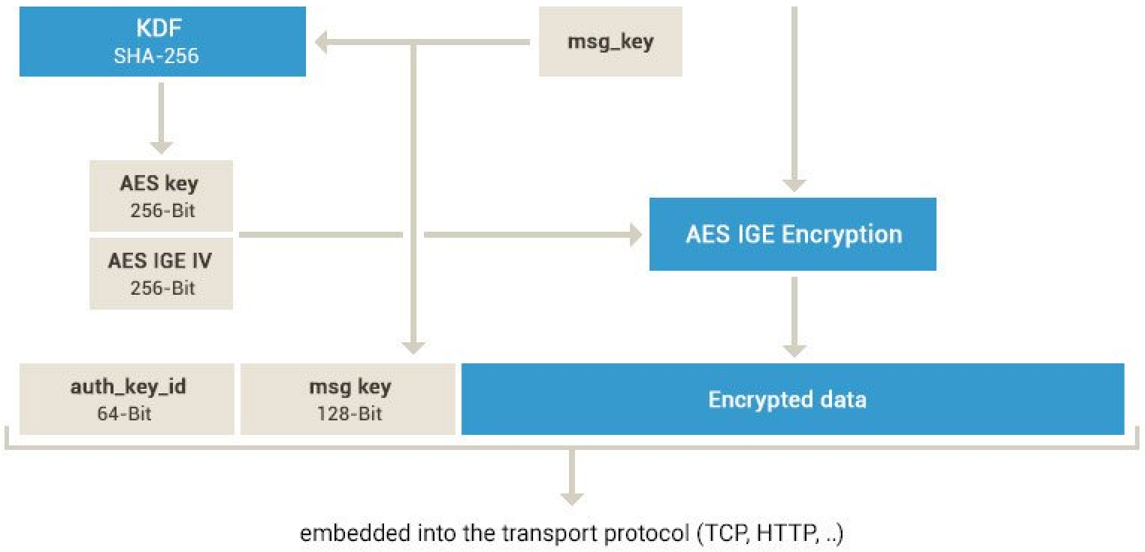
Protocol description

Before a message (or a multipart message) is transmitted over a network using a transport protocol, it is encrypted in a certain way, and an external header is added at the top of the message that consists of a 64-bit key identifier auth_key_id (that uniquely identifies an authorization key for the server as well as the user) and a 128-bit message key msg_key.

The authorization key auth_key combined with the message key msg_key define an actual 256-bit key aes_key and a 256-bit initialization vector aes_iv, which are used to encrypt the message using AES-256 encryption in infinite garble extension (IGE) mode. Note that the initial part of the message to be encrypted contains variable data (session, message ID, sequence number, server salt) that obviously influences the message key (and thus the AES key and iv). In MTProto 2.0, the message key is defined as the 128 middle bits of the SHA-256 of the message body (including session, message ID, padding, etc.) prepended by 32 bytes taken from the authorization key. In the older MTProto 1.0, the message key was computed as the lower 128 bits of SHA-1 of the message body, excluding the padding bytes.

Multipart messages are encrypted as a single message.


MTProto 2.0, part I
Cloud chats (server-client encryption)
to be encrypted
shared key (auth_key)
persistent, generated via DH
Salt
64-Bit
Session_id
64-Bit
Payload *
Padding
12-1024 bytes
SHA-256
Note:
Payload always contains time, length and sequence number to be checked by the receiving party after decryption.



Important: After decryption, the receiver must check that
msg_key = SHA-256(fragment of auth_key + decrypted data)

> Got questions about this setup? — Check out the Advanced FAQ!

Note 1

Each plaintext message to be encrypted in MTProto always contains the following data to be checked upon decryption in order to make the system robust against known problems with the components:

- server salt (64-Bit)
- session id
- message sequence number
- message length
- time

Note 2

Telegram's End-to-end encrypted Secret Chats are using an additional layer of encryption on top of the described above. See Secret Chats, End-to-End encryption for details.

Terminology

Authorization Key (auth_key)

A 2048-bit key shared by the client device and the server, created upon user registration directly on the client device by exchanging Diffie-Hellman keys, and never transmitted over a network. Each authorization key is user-specific. There is nothing that prevents a user from having several keys (that correspond to “permanent sessions” on different devices), and some of these may be locked forever in the event the device is lost. See also Creating an Authorization Key.

Server Key

A 2048-bit RSA key used by the server digitally to sign its own messages while registration is underway and the authorization key is being generated. The application has a built-in public server key which can be used to verify a signature but cannot be used to sign messages. A private server key is stored on the server and changed very infrequently.

Key Identifier (auth_key_id)

The 64 lower-order bits of the SHA1 hash of the authorization key are used to indicate which particular key was used to encrypt a message. Keys must be uniquely defined by the 64 lower-order bits of their SHA1, and in the event of a collision, an authorization key is regenerated. A zero key identifier means that encryption is not used which is permissible for a limited set of message types used during registration to generate an authorization key in a Diffie-Hellman exchange. For MTProto 2.0, SHA1 is still used here, because auth_key_id should identify the authorization key used independently of the protocol version.

Session

A (random) 64-bit number generated by the client to distinguish between individual sessions (for example, between different instances of the application, created with the same authorization key). The session in conjunction with the key identifier corresponds to an application instance. The server can maintain session state. Under no circumstances can a message meant for one session be sent into a different session. The server may unilaterally forget any client sessions; clients should be able to handle this.

Server Salt

A (random) 64-bit number periodically (say, every 24 hours) changed (separately for each session) at the request of the server. All subsequent messages must contain the new salt (although, messages with the old salt are still accepted for a further 300 seconds). Required to protect against replay attacks and certain tricks associated with adjusting the client clock to a moment in the distant future.

Message Identifier (msg_id)

A (time-dependent) 64-bit number used uniquely to identify a message within a session. Client message identifiers are divisible by 4, server message identifiers modulo 4 yield 1 if the message is a response to a client message, and 3 otherwise. Client message identifiers must increase monotonically (within a single session), the same as server message identifiers, and must approximately equal unixtime*2^32. This way, a message identifier points to the approximate moment in time the message was created. A message is rejected over 300 seconds after it is created or 30 seconds before it is created (this is needed to protect from replay attacks). In this situation, it must be re-sent with a different identifier (or placed in a container with a higher identifier). The identifier of a message container must be strictly greater than those of its nested messages.

Important: to counter replay-attacks the lower 32 bits of msg_id passed by the client must not be empty and must present a fractional part of the time point when the message was created.

Content-related Message

A message requiring an explicit acknowledgment. These include all the user and many service messages, virtually all with the exception of containers and acknowledgments.

Message Sequence Number (msg_seqno)

A 32-bit number equal to twice the number of “content-related” messages (those requiring acknowledgment, and in particular those that are not containers) created by the sender prior to this message and subsequently incremented by one if the current message is a content-related message. A container is always generated after its entire contents; therefore, its sequence number is greater than or equal to the sequence numbers of the messages contained in it.

Message Key (msg_key)

In MTProto 2.0, the middle 128 bits of the SHA-256 hash of the message to be encrypted (including the internal header and the padding bytes for MTProto 2.0), prepended by a 32-byte fragment of the authorization key.

In MTProto 1.0, message key was defined differently, as the lower 128 bits of the SHA-1 hash of the message to be encrypted, with padding bytes excluded from the computation of the hash. Authorization key was not involved in this computation.

Internal (cryptographic) Header

A header (16 bytes) added before a message or a container before it is all encrypted together. Consists of the server salt (64 bits) and the session (64 bits).

External (cryptographic) Header

A header (24 bytes) added before an encrypted message or a container. Consists of the key identifier auth_key_id (64 bits) and the message key msg_key (128 bits).

Payload

External header + encrypted message or container.

Defining AES Key and Initialization Vector

The 2048-bit authorization key (auth_key) and the 128-bit message key (msg_key) are used to compute a 256-bit AES key (aes_key) and a 256-bit initialization vector (aes_iv) which are subsequently used to encrypt the part of the message to be encrypted (i. e. everything with the exception of the external header that is added later) with AES-256 in infinite garble extension (IGE) mode.

For MTProto 2.0, the algorithm for computing aes_key and aes_iv from auth_key and msg_key is as follows.

- msg_key_large = SHA256 (substr (auth_key, 88+x, 32) + plaintext + random_padding);
- msg_key = substr (msg_key_large, 8, 16);
- sha256_a = SHA256 (msg_key + substr (auth_key, x, 36));
- sha256_b = SHA256 (substr (auth_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);

where x = 0 for messages from client to server and x = 8 for those from server to client.

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

The lower-order 1024 bits of auth_key are not involved in the computation. They may (together with the remaining bits or separately) be used on the client device to encrypt the local copy of the data received from the server. The 512 lower-order bits of auth_key are not stored on the server; therefore, if the client device uses them to encrypt local data and the user loses the key or the password, data decryption of local data is impossible (even if data from the server could be obtained).

In MTProto 1.0, when AES was used to encrypt a block of data of a length not divisible by 16 bytes, the data was padded with 0 to 15 random padding bytes random_padding to a length divisible by 16 bytes prior to encryption. In MTProto 2.0, this padding is taken into account when computing `msg_key`. Note that MTProto 2.0 requires from 12 to 1024 bytes of padding, still subject to the condition that the resulting message length be divisible by 16 bytes.

Using MTProto 2.0 instead of MTProto 1.0

A client may either use only MTProto 2.0 or only MTProto 1.0 in the same TCP connection. The server detects the protocol used by the first message received from the client, and then uses the same encryption for its messages, and expects the client to use the same encryption henceforth. We recommend using MTProto 2.0; MTProto 1.0 is deprecated and supported for backward compatibility only.

Important Checks

When an encrypted message is received, it must be checked that msg_key is in fact equal to the 128 middle bits of the SHA-256 of the decrypted data with a 32-byte fragment of auth_key prepended to it, and that msg_id has even parity for messages from client to server, and odd parity for messages from server to client.

In addition, the identifiers (msg_id) of the last N messages received from the other side must be stored, and if a message comes in with msg_id lower than all or equal to any of the stored values, the message is to be ignored. Otherwise, the new message msg_id is added to the set, and, if the number of stored msg_id values is greater than N, the oldest (i. e. the lowest) is forgotten.

On top of this, msg_id values that belong over 30 seconds in the future or over 300 seconds in the past are to be ignored. This is especially important for the server. The client would also find this useful (to protect from a replay attack), but only if it is certain of its time (for example, if its time has been synchronized with that of the server).

Certain client-to-server service messages containing data sent by the client to the server (for example, msg_id of a recent client query) may, nonetheless, be processed on the client even if the time appears to be “incorrect”. This is especially true of messages to change server_salt and notifications of invalid client time. See Mobile Protocol: Service Messages.

Storing an Authorization Key on a Client Device

It may be suggested to users concerned with security that they password protect the authorization key in approximately the same way as in ssh. This can be accomplished by prepending the value of cryptographic hash function, such as SHA-256, of the key to the front of the key, following which the entire string is encrypted using AES in CBC mode and a key equal to the user’s (text) password. When the user inputs the password, the stored protected password is decrypted and verified by checking the SHA-256 value. From the user’s standpoint, this is practically the same as using an application or a website password.

Unencrypted Messages

Special plain-text messages may be used to create an authorization key as well as to perform a time synchronization. They begin with auth_key_id = 0 (64 bits) which means that there is no auth_key. This is followed directly by the message body in serialized format without internal or external headers. A message identifier (64 bits) and body length in bytes (32 bytes) are added before the message body.

Only a very limited number of messages of special types can be transmitted as plain text.

Schematic Presentation of Messages

Encrypted Message

auth_key_id	msg_key	encrypted_data
int64	int128	bytes

Encrypted Message: encrypted_data

Contains the cypher text for the following data:

salt	session_id	message_id	seq_no	message_data_length	message_data	padding12..1024
int64	int64	int64	int32	int32	bytes	bytes

Unencrypted Message

auth_key_id = `0`	message_id	message_data_length	message_data
int64	int64	int32	bytes

MTProto 2.0 uses 12..1024 padding bytes, instead of the 0..15 used in MTProto 1.0

Creating an Authorization Key

An authorization key is normally created once for every user during the application installation process immediately prior to registration. Registration itself, in actuality, occurs after the authorization key is created. However, a user may be prompted to complete the registration form while the authorization key is being generated in the background. Intervals between user key strokes may be used as a source of entropy in the generation of high-quality random numbers required for the creation of an authorization key.

See Creating an Authorization Key.

During the creation of the authorization key, the client obtains its server salt (to be used with the new key for all communication in the near future). The client then creates an encrypted session using the newly generated key, and subsequent communication occurs within that session (including the transmission of the user's registration information and phone number validation) unless the client creates a new session. The client is free to create new or additional sessions at any time by choosing a new random session_id.