MTProto Mobile Protocol

> Please feel free to check out our FAQ for the Technically Inclined. Client developers are required to comply with the Security Guidelines.

Related articles

Mobile Protocol: Detailed Description

Creating an Authorization Key

Creating an Authorization Key: Example

Mobile Protocol: Service Messages

Mobile Protocol: Service Messages about Messages

Binary Data Serialization

TL Language

MTProto TL-schema

End-to-end encryption, Secret Chats

End-to-end TL-schema

Security Guidelines for Client Software Developers

This page deals with the basic layer of MTProto encryption used for Cloud chats (server-client encryption). See also:

- Secret Chats, end-to-end-encryption
- End-to-end encrypted Voice Calls

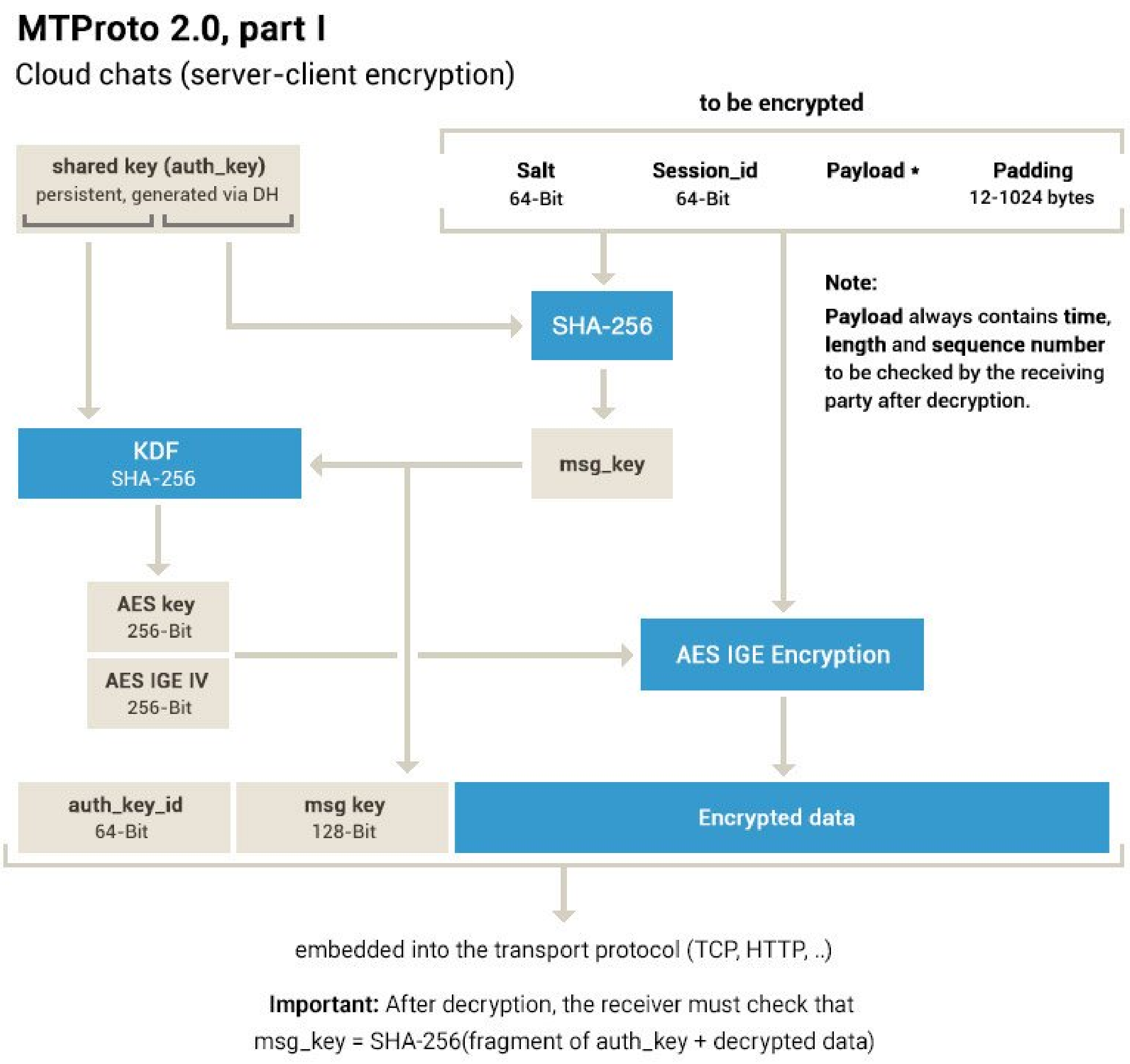
General Description

The protocol is designed for access to a server API from applications running on mobile devices. It must be emphasized that a web browser is not such an application.

The protocol is subdivided into three virtually independent components:

- High-level component (API query language): defines the method whereby API queries and responses are converted to binary messages.
- Cryptographic (authorization) layer: defines the method by which messages are encrypted prior to being transmitted through the transport protocol.
- Transport component: defines the method for the client and the server to transmit messages

over some other existing network protocol (such as HTTP, HTTPS, WS (plain websockets), WSS (websockets over HTTPS), TCP, UDP).


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.
KDF
SHA-256
msg_key
AES key
256-Bit
AES IGE IV
256-Bit
AES IGE Encryption
auth_key_id
64-Bit
msg key
128-Bit
Encrypted data
embedded into the transport protocol (TCP, HTTP, ..)
Important: After decryption, the receiver must check that
msg_key = SHA-256(fragment of auth_key + decrypted data)

> As of version 4.6, major Telegram clients are using MTProto 2.0, described in this article. MTProto v1.0 (described here for reference) is deprecated and is currently being phased out.

Brief Component Summary

High-Level Component (RPC Query Language/API)

From the standpoint of the high-level component, the client and the server exchange messages inside a session. The session is attached to the client device (the application, to be more exact) rather than a specific websocket/http/https/tcp connection. In addition, each session is attached to a user key ID by which authorization is actually accomplished.

Several connections to a server may be open; messages may be sent in either direction through any of the connections (a response to a query is not necessarily returned through the same connection that carried the original query, although most often, that is the case; however, in no case can a message be returned through a connection belonging to a different session). When the UDP protocol is used, a response might be returned by a different IP address than the one to which the query had been sent.

There are several types of messages:

- RPC calls (client to server): calls to API methods
- RPC responses (server to client): results of RPC calls
- Message received acknowledgment (or rather, notification of status of a set of messages)
- Message status query
- Multipart message or container (a container that holds several messages; needed to send several RPC calls at once over an HTTP connection, for example; also, a container may support gzip).

From the standpoint of lower level protocols, a message is a binary data stream aligned along a 4 or 16-byte boundary. The first several fields in the message are fixed and are used by the cryptographic/authorization system.

Each message, either individual or inside a container, consists of a message identifier (64 bits, see below), a message sequence number within a session (32 bits), the length (of the message body in bytes; 32 bits), and a body (any size which is a multiple of 4 bytes). In addition, when a container or a single message is sent, an internal header is added at the top (see below), then the entire message is encrypted, and an external header is placed at the top of the message (a 64-bit key identifier and a 128-bit message key).

A message body normally consists of a 32-bit message type followed by type-dependent parameters. In particular, each RPC function has a corresponding message type. For more detail, see Binary Data Serialization, Mobile Protocol: Service Messages.

All numbers are written as little endian. However, very large numbers (2048-bit) used in RSA and DH are written in the big endian format because that is how the OpenSSL library does it.

Authorization and Encryption

Prior to a message (or a multipart message) being 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 which is: a 64-bit key identifier (that uniquely identifies an authorization key for the server as well as the user) and a 128-bit message key. A user key together with the message key defines an actual 256-bit key which is what encrypts the message using AES-256 encryption. 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). The message key is defined as the 128 middle bits of the SHA256 of the message body (including session, message ID, etc.), including the padding bytes, prepended by 32 bytes taken from the authorization key. Multipart messages are encrypted as a single message.

> For a technical specification, see Mobile Protocol: Detailed Description

The first thing a client application must do is create an authorization key which is normally generated when it is first run and almost never changes.

The protocol's principal drawback is that an intruder passively intercepting messages and then somehow appropriating the authorization key (for example, by stealing a device) will be able to decrypt all the intercepted messages post factum. This probably is not too much of a problem (by stealing a device, one could also gain access to all the information cached on the device without decrypting anything); however, the following steps could be taken to overcome this weakness:

- Session keys generated using the Diffie–Hellman protocol and used in conjunction with the authorization and the message keys to select AES parameters. To create these, the first thing a client must do after creating a new session is send a special RPC query to the server (“generate session key”) to which the server will respond, whereupon all subsequent messages within the session are encrypted using the session key as well.
- Protecting the key stored on the client device with a (text) password; this password is never stored in memory and is entered by a user when starting the application or more frequently (depending on application settings).
- Data stored (cached) on the user device can also be protected by encryption using an authorization key which, in turn, is to be password-protected. Then, a password will be required to gain access even to that data.

Time Synchronization

If client time diverges widely from server time, a server may start ignoring client messages, or vice versa, because of an invalid message identifier (which is closely related to creation time). Under these circumstances, the server will send the client a special message containing the correct time and a certain 128-bit salt (either explicitly provided by the client in a special RPC synchronization request or equal to the key of the latest message received from the client during the current session). This message could be the first one in a container that includes other messages (if the time discrepancy is significant but does not as yet result in the client’s messages being ignored).

Having received such a message or a container holding it, the client first performs a time synchronization (in effect, simply storing the difference between the server’s time and its own to be able to compute the “correct” time in the future) and then verifies that the message identifiers for correctness.

Where a correction has been neglected, the client will have to generate a new session to assure the monotonicity of message identifiers.

MTProto transport

Before being sent using the selected transport protocol, the payload has to be wrapped in a secondary protocol header, defined by the appropriate MTProto transport protocol.

- Abridged
- Intermediate
- Padded intermediate
- Full

The server recognizes these different protocols (and distinguishes them from HTTP, too) by the header. Additionally, the following transport features can be used:

- Quick ack
- Transport errors
- Transport obfuscation

Example implementations for these protocols can be seen in tdlib and MadelineProto.

Transport

Enables the delivery of encrypted containers together with the external header (hereinafter, Payload) from client to server and back. Multiple transport protocols are defined:

- TCP
- Websocket
- Websocket over HTTPS
- HTTP
- HTTPS
- UDP

(We shall examine only the first five types.)

Recap

To recap, using the ISO/OSI stack as comparison:

- Layer 7 (Application): High-level RPC API
- Layer 6 (Presentation): Type Language
- Layer 5 (Session): MTProto session
- Layer 4 (Transport):
    - 4.3: MTProto transport protocol
    - 4.2: MTProto obfuscation (optional)
    - 4.1: Transport protocol
- Layer 3 (Network): IP
- Layer 2 (Data link): MAC/LLC
- Layer 1 (Physical): IEEE 802.3, IEEE 802.11, etc...