UDC 004.056.55

MTPROTO PROTOCOL FOR ENCRYPTING MOBILE CLOUD CHAT APPLICATIONS

MEDYUHO DMITRY, TATYANA RUDKOVA
Polotsk State University, Belarus

The protocol is designed for access to a server API from applications running on mobile devices. This symmetric encryption scheme is based on 256-bit symmetric AES encryption, 2048-bit RSA encryption and Diffie–Hellman key exchange.

The protocol is subdivided into three virtually independent components:

1. *High-level component* (API query language): defines the method whereby API queries and responses are converted to binary *messages*.
2. *Cryptographic* (authorization) layer: defines the method by which messages are encrypted prior to being transmitted through the transport protocol.
3. *Transport component*: defines the method for the client and the server to transmit messages over some other existing network protocol (such as HTTP, HTTPS, TCP, UDP) [1].

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 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:

1. RPC calls (client to server): calls to API methods.
2. RPC responses (server to client): results of RPC calls.
3. Message received acknowledgment (or rather, notification of status of a set of messages).
4. Message status query.
5. 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), 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, 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*).

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 [2]. 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.

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 ac-

cess to all the information cached on the device without decrypting anything); however, the following steps could be taken to overcome this weakness:

1. Session keys generated using the Diffie-Hellman protocol and used together 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 to 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 [3].
2. 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).
3. 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 an access even to those data.

HTTP Transport

Implemented over HTTP/1.1 (with keepalive) running over the traditional TCP Port 80. HTTPS can also be used in addition to the encryption method described above.

An HTTP connection is attached to a session (or rather, to session + key identifier) specified in the most recent user query received; normally, the session is the same in all queries, but crafty HTTP proxies may corrupt that. A server may not return a message into an HTTP connection unless it belongs to the same session, and unless it is the server's turn (an HTTP request had been received from the client to whom a response has not been sent yet) [4].

The overall arrangement is as follows. The client opens one or more keepalive HTTP or HTTPS connections to the server. If one or more messages need to be sent, they are made into a *payload* which is followed by a POST request to the URL/api to which the payload is transmitted as data. In addition, *Content-Length*, *Keepalive*, and *Host* are valid HTTP headers.

Having received the query, the server may either wait a little while (if the query requires a response following a short timeout) or immediately return a dummy response (only acknowledging the receipt of the container). In any case, the response may contain any number of messages. The server may at the same time send out any other messages it might be holding for the session.

In addition, there exists a special long poll RPC query (valid for HTTP connections only) which transmits maximum timeout *T*. If the server has messages for the session, they are returned immediately; otherwise, a wait state is entered until such time as the server has a message for the client or *T* seconds have elapsed. If no events occur in the span of *T* seconds, a dummy response is returned (special message).

If a server needs to send a message to a client, it checks for an HTTP connection that belongs to the required session and is in the "answering an HTTP request" state (including long poll) whereupon the message is added to the response container for the connection and sent to the user. In a typical case, there is some additional wait time (50 milliseconds) against the eventuality that the server will soon have more messages for the session.

If no suitable HTTP connection is available, the messages are placed in the current session's send queue. However, they find their way there anyway until receipt is explicitly confirmed by the client. For all protocols, the client must return an explicit acknowledgment within a reasonable time (it can be added to a container for the following request).

If the sent queue overflows or if messages stay in the queue for over 10 minutes, the server forgets them. This may happen even faster, if the server is running out of buffer space (for example, because of serious network issues resulting in a large number of connections becoming severed).

TCP Transport

This is very similar to the HTTP transport. It may also be implemented over Port 80 (to penetrate all firewalls) and even use the same server IP addresses. In this situation, the server understands whether HTTP or TCP protocol must be used for the connection, based on the first four incoming bytes (for HTTP, it is POST).

When a TCP connection is created, it is assigned to the session (and the authorization key) transmitted in the first user's message, and subsequently used exclusively for this session (multiplexing arrangements are not allowed).

If a payload (packet) needs to be transmitted from the server to a client or from a client to the the server, it is encapsulated as follows: 4 length bytes are added at the front (to include the length, the sequence number, and CRC32; always divisible by 4) and 4 bytes with the packet sequence number within this TCP connection (the first packet sent is numbered 0, the next one 1, etc.), and 4 CRC32 bytes at the end (length, sequence number, and payload together).

There is an *abridged* version of the same protocol: if the client sends *0xef* as the first byte (only prior to the very first data packet), then packet length is encoded by a single byte (*0x01..0x7e* = data length divided by 4; or *0x7f* followed by 3 length bytes (little endian) divided by 4) followed by the data themselves (sequence number and CRC32 not added). In this case, the server responses look the same (the server does not send *0xef* as the first byte).

In case 4-byte data alignment is needed, an *intermediate* version of the original protocol may be used: if the client sends *0xeeeeeeee* as the first int (four bytes), then packet length is encoded always by four bytes as in the original version, but the sequence number and CRC32 are omitted, thus decreasing total packet size by 8 bytes.

The full, the intermediate and the abridged versions of the protocol have support for quick acknowledgment. In this case, the client sets the highest-order length bit in the query packet, and the server responds with a special 4 bytes as a separate packet. They are the 32 higher-order bits of SHA256 of the encrypted portion of the packet prepended by 32 bytes from the authorization key (the same hash as computed for verifying the message key), with the most significant bit set to make clear that this is not the length of a regular server response packet; if the abridged version is used, bswap is applied to these four bytes.

There are no implicit acknowledgments for the TCP transport: all messages must be acknowledged explicitly. Most frequently, acknowledgments are placed in a container with the next query or response if it is transmitted in short order. For example, this is almost always the case for a client's messages containing RPC queries: the acknowledgment normally arrives with the RPC response.

In the case of an error, the server may send a packet with a payload of 4 bytes, containing the error code. For example, Error Code 403 corresponds to situations where the corresponding HTTP error would have been returned by the HTTP protocol [5].

REFERENCES

1. List of network protocols [Electronic resource]. – Access mode: https://en.wikipedia.org/wiki/ Lists_of_network_protocols. – Access date: 10.02.2018.
2. Advanced Encryption Standart [Electronic resource]. – Access mode: https://ru.wikipedia.org/wiki/ Advanced_Encryption_Standard. – Access date: 12.02.2018.
3. Diffie–Hellman key exchange [Electronic resource]. – Access mode: https://en.wikipedia.org/wiki/ Diffie%E2%80%93_Hellman_key_exchange. – Access date: 13.02.2018.
4. HyperText Transfer Protocol [Electronic resource]. – Access mode: https://en.wikipedia.org/wiki/ Hypertext_Transfer_Protocol. – Access date: 14.02.2018.
5. List of HTTP status codes [Electronic resource]. – Access mode: https://en.wikipedia.org/wiki/ List_of_HTTP_status_codes. – Access date: 14.02.2018.