Threema protocol analysis

Jan Ahrens

E-mail jan.ahrens@gmail.com
PGP key 3762 1152 E099 AB27 04E8
3FD1 B911 E6A2 2B4F 3B5F

2014-03-22

Abstract

Threema is a commercial mobile messaging application developed by the Swiss-based company Kasper Systems GmbH. Its popularity is based on the promise that each message is transferred using end-to-end encryption. While claiming to use the open-source cryptographic library NaCl, the details of the used protocol are closed and can not be independently reviewed and verified. The Validation Logging functionality provided by Threema does not prove that logged messages are exchanged with the server.

In this paper I will describe the Threema protocol, with the intention of enabling an independent review. This paper does not judge whether there are weaknesses in the protocol or the application itself.

The results are based on the implementation used by the Threema Android application version 1.3. Neither its keys nor servers are included in this paper. They are sold by Kasper System GmbH in form of the Threema binary.

Contents

1 Introduction

Threema is a commercial mobile messaging application for Android and iOS. Its features include chatting with contacts, group conversations and sharing images. All communication is said to be end-to-end encrypted using a keypair generated by the application on its first launch.

Threema is sold by the Swiss-based company Kasper Systems GmbH on its website, through the Google Play Store and the Apple App Store. Threema's source code is closed and can not be reviewed without the agreement of its authors. The application includes a Validation Logging feature that can be used to log the in- and outgoing messages. This feature logs the message bodies, but not all the data that is exchanged with the server. There is no way to prove that the message that is being logged correlates to data that is being sent by the application. To enable an independent review I will describe the details of the custom protocol, that Threema is using, in this paper. The results are based on an analysis of the Threema Android application version 1.3.

This paper begins with an introduction of the used cryptography library in Section 2 and continues with an explanation of the used keys in Section 3. Before data can be exchanged with the Threema server, a connection has to be established. This handshake will be described in Section 4. After a successful handshake server and client can exchange data. The different kinds of messages are discussed in Section 5. Section 6 will give some advice on how to verify the described protocol by looking at the traffic generated by the application.

2 The NaCl library

The protocol used by Threema is based on the NaCl library. On its project page NaCl is being described as:

> NaCl (pronounced "salt") is a new easy-to-use high-speed software library for network communication, encryption, decryption, signatures, etc. NaCl's goal is to provide all of the core operations needed to build higher-level cryptographic tools.

NaCl provides the *crypto_box* function, among other primitives, that can be used to exchange authenticated messages between a sender and a recipient. Given a message m, the ciphertext c is produced by using the recipient's public key PK_r, the senders secret key SK_s and a nonce n.

$$c = crypto\_box(m, n, PK_r, SK_s) \tag{1}$$

The recipient can decipher this message using the *crypto_box_open* function with his secret key SK_r, the senders public key PK_s and the nonce n used to create the ciphertext.

$$m = crypto\_box\_open(c, n, PK_s, SK_r) \tag{2}$$

In contrast to traditional public-key cryptosystems, like RSA, a ciphertext generated using the *crypto_box* function can not only be deciphered with the recipient's secret key. It can also be deciphered using the secret key of the sender. This property is the result of *crypto_box* using elliptic curve Diffie-Hellmann internally to derive a shared secret between both keypairs.

3 Short- and long-term keys

Each Threema client generates a keypair on its first launch. This keypair is called the long-term keypair and consists of a public key LPK_c and a secret key LSK_c. The long-term public key is linked to an 8 byte username by sending it to the Threema server[1].

The long-term public key has to be exchanged with other Threema users. To send a message the client has to know the username and associated long-term public key of the recipient. The details will be discussed in Section 5.2.

Whenever a Threema client wants to establish a connection with the server it will generate another keypair, used only for a short amount of time. This keypair is called the short-term keypair and consists of the short-term public key SPK_c and the short-term secret key SSK_s. The details of this handshake are explained in the next section.

4 Handshake

The protocol is based on TCP using the client-server model. On top of a TCP handshake, the client has to perform a cryptographic handshake with the server before it can send messages. This handshake is similar to the one in the CurveCP protocol. During the handshake, client and server will exchange, and agree on, the following data:

- Client short-term public key: $SPK_c \in \{0..255\}^{32}$
- Server short-term public key: $SPK_s \in \{0..255\}^{32}$
- Client nonce prefix: $NP_c \in \{0..255\}^{16}$
- Server nonce prefix: $NP_s \in \{0..255\}^{16}$
- Client username: $u \in \{A..Z, 0..9\}^8$
- Client system data, such as the Threema version, platform and the Java runtime

The nonce prefix is used together with an 8 byte counter to generate a unique 24 byte nonce for every message[2]. I will be using the notation $n_{xi} = nonce(NP_x, i)$ to express that the nonce n_{xi} is based on the nonce prefix NP_x and the counter value i.

[1] This is done using the Threema REST API. The REST API is not a direct part of the communication protocol and will not be discussed in this paper.

[2] This method is explained in this Stack Overflow discussion.

4.1 Client Hello

The connection to the server is initialized by the client sending the Client Hello packet (Figure 1). Before sending this packet, the client generates a short-term keypair using the crypto_box_keypair function from the NaCl library. It will also generate a 16 byte random nonce prefix NP_c, that is used to generate unique nonces later on.

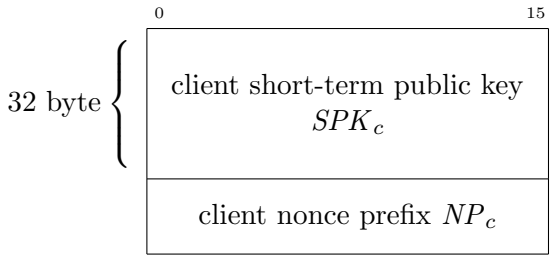

Figure 1: The Client Hello packet

4.2 Server Hello

After receiving the Client Hello, the server will respond with a Server Hello packet (Figure 2). Before sending this packet, it also generates a short-term keypair and a 16 byte random nonce prefix NP_s.

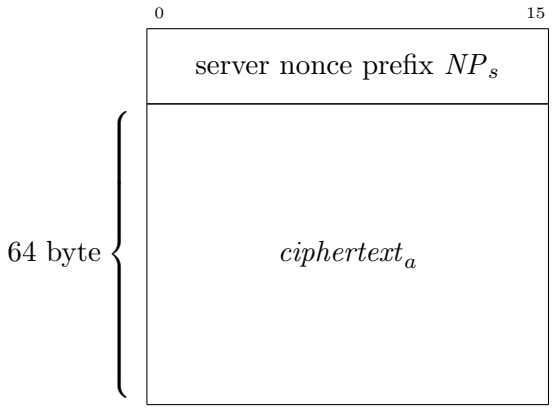

Figure 2: The Server Hello packet

Along with the nonce prefix NP_s the server will send the $ciphertext_a$. The contents of that ciphertext are the server's short-term public key SPK_s and the client's nonce prefix NP_c. The client's nonce prefix is included to confirm that the server received to correct one. Because the client does not yet know about the server's short-term key, the server's long-term secret key has to be used to create the ciphertext (Equation 3).

$$ciphertext_a = crypto\_box(SPK_s + NP_c, nonce(NP_s, 1), SPK_c, LSK_s) \tag{3}$$

4.3 Authentication

When the server has introduced itself with the Server Hello packet, the client sends the encrypted authentication packet (Figure 3 and Figure 4). Its purpose is to authenticate the user, confirm the server's nonce prefix and send some data about the client's system, like the used Threema version. The authentication packet is the first packet that is being encrypted using both short-term keys. (Equation 4).

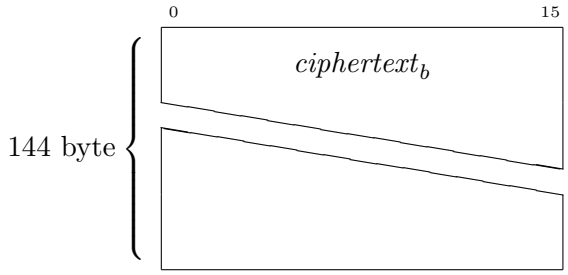

Figure 3: The Authentication packet. It is fully encrypted with the short-term keys.

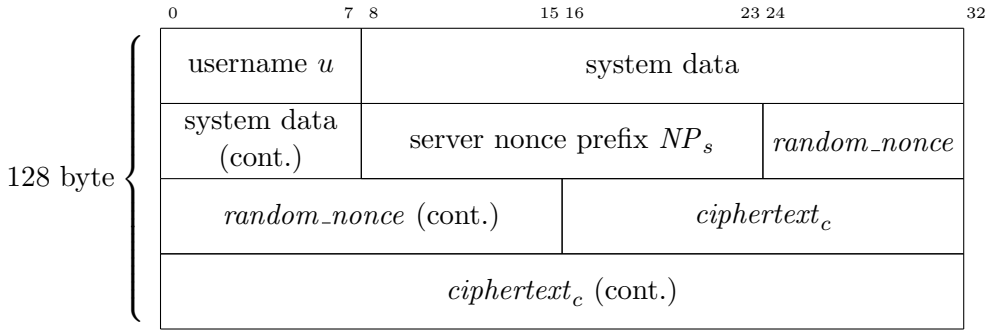

Figure 4: Structure of the *authentication_data* that is being encrypted as $ciphertext_b$

$$ciphertext_b = crypto\_box(authentication\_data, nonce(NP_c, 1), SPK_s, SSK_c) \tag{4}$$

With the $ciphertext_c$ inside *authentication_data* (Equation 5) the client is verifying that it possesses the private key LSK_c that belongs to the long-term public key LPK_c and is linked to the username u.

$$ciphertext_c = crypto\_box(SPK_c, random\_nonce, LPK_s, LSK_c) \tag{5}$$

4.4 Acknowledgment

In the last step of the handshake, the server acknowledges that the authentication data provided by the client is valid. The 32 byte packet is fully encrypted and can be decrypted as shown in equation 6. Its content is meaningless and contains only zeros. When the packet can be decrypted, the handshake is completed and a connection with the Threema server has been established.

$$message_d = crypto\_box\_open(ciphertext_d, nonce(NP_s, 2), SPK_s, SSK_c) \tag{6}$$

5 Exchanging data with the server

After the handshake is completed, the client and the server are allowed to exchange data. The data is structured into different kinds of *data packets* that will be described in the next section. Whenever a number is used in any of the data packets, it will be encoded using the little-endian byte order. For example the unsigned short integer 2342_{10} is encoded as 0926_{16} using big-endian and as 2609_{16} using little-endian. Since big-endian is the standard format on most client platforms, numbers have to be converted between little- and big-endian in most of the cases.

5.1 General format of a data packet

A data packet is the general format for data exchanged between the server and the client. Each data packet is prefixed with a two byte length field. It contains an unsigned short integer encoded as little-endian. After reading those two bytes, the receiver knows how many bytes it has to read from the sender in total.

Each data packet is encrypted and can be decrypted by using the appropriate short-term keys for the client and server. Depending on who sent packet, the next nonce generated by the server's or the client's nonce prefix has to be used.

Type	Description
`0x01`	Sending message
`0x02`	Delivering message
`0x81`	Server Acknowledgment
`0x82`	Client Acknowledgment
`0xd0`	Connection established

Table 1: Data packet types

The first four bytes of the decrypted data packet are used to encode its type. Table 1 shows some of the defined types. Most data packets will contain additional information besides its type identifier. One exception is the data packet `0xd0`. It will be sent by the server after the connection is successfully established and all queued messages have been sent to the client. It is used by the client to know when the user can start sending messages. The meaning of the remaining data packet types will be explained in the following section.

5.2 Sending and receiving messages

The core purpose of the Threema protocol is to exchange messages with other users. Data packets with the type `0x01` and `0x02` are used for this (Figure 7). The difference between the two types is that `0x01` is used for outgoing and `0x02` is used for incoming messages. Messages are used for different purposes in the Threema protocol. In order to differentiate between the various kinds of messages, each message starts with a one byte message type identifier. Table 2 lists some of them.

Type	Description
`0x01`	Text message
`0x80`	Message delivery receipt
`0x90`	User typing notification

Table 2: Message types used inside the data packet `0x01` and `0x02`

Each Threema message has a unique identifier in order to be referenced by the server and by the recipient. Figure 5 shows an example of a message exchange between two users and the server: After a user composed a message it will be sent to the server. The server will acknowledge that it has received the message. When the message has been delivered to the recipient, its delivery will be acknowledged, by generating a new message containing the delivery receipt (`0x80`).

Acknowledgments are data packets sent after the client or server have received a message packet. Each acknowledgment packet includes the related message id and the sender of the message. The identifier `0x81` is used by the server to acknowledge that it has queued a message. The client uses the identifier `0x82` to acknowledge that is has received one. If the server does not receive an acknowledgment, it will attempt to re-transmit the message. The same applies

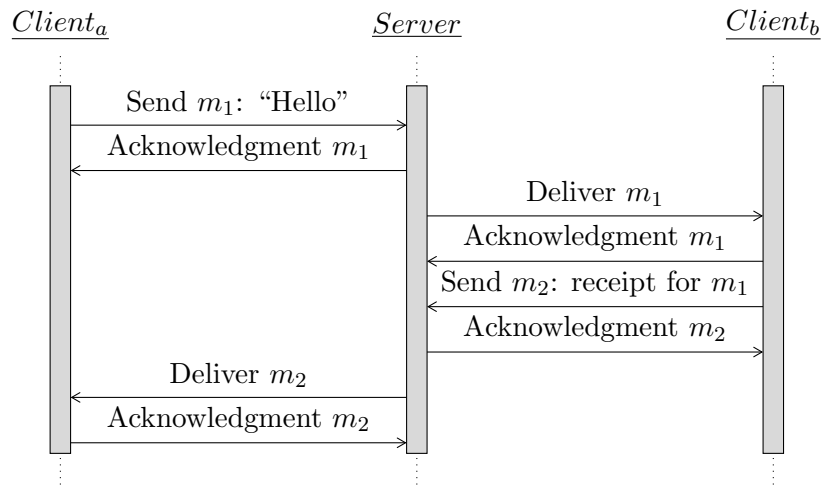

Figure 5: Sending a message (m_1) and confirming its delivery by the recipient (m_2)

to the client if the server does not acknowledge the queuing of a message. Figure 6 shows the structure of an acknowledgment packet.

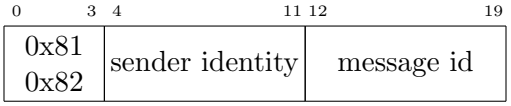

Figure 6: Acknowledgment packet

The structure of the message packet is show in Figure 7. The client will used `0x01` as a message identifier to send a message and the server will use `0x02` to deliver a message. Sender and recipient are filled with the 8 byte identity of the related user. The message id is randomly generated. The *time* field encodes the current UTC time as an unsigned integer in little-endian byte order. The meaning of the four bytes in the *dunno* field is not yet known. It contains zeros in most cases. A fully random nonce will be used to encrypt the actual message content.

The server might know about the contents of a message by looking at its length. To prevent this each message will be extended using PKCS7 padding before its encrypted. For example the “user starts typing” message has always the same length of four bytes (9000). Using PKCS7 padding with a random number of padding bytes (i.e. 5 bytes), the message becomes longer (90000505050505) and the content is no longer guessable by looking at its ciphertext.

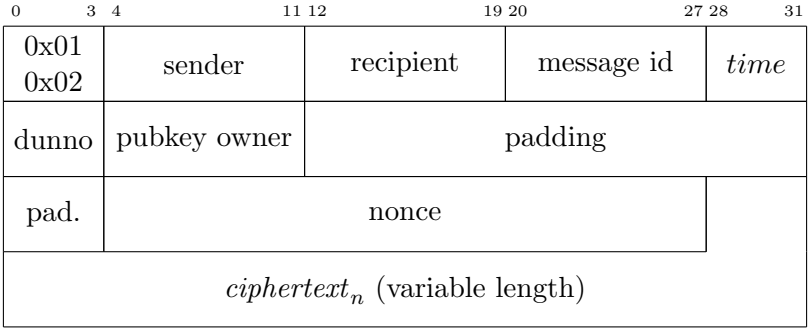

Figure 7: A message packet

6 Decrypt client-server communication

If you want to verify the protocol described in this paper, you need to know the client's short-term secret key SSK_c, the server's short-term public key SPK_s as well as the nonce prefixes NP_c and NP_s. In order to calculate the counter values used to generate nonces by the client and the server you either need to have all exchanged packets or guess the correct value by using brute force.

If you have all exchanged packets, another way of deciphering the transport encryption is to use the server's long-term public key LPK_s together with the client's short-term secret key SSK_c. Using both keys it is possible to decipher the whole handshake and extract all session parameters.

The method used to gain these keys during this analysis was to modify the app and include additional logging output.

Once you removed the transport encryption used between client and server you can read the payload of every message (see section 5.2 for an example). The Threema Validation Logging feature and the related programs can be used to compare the logged data and the intercepted message contents then.

7 Summary

I did not describe the protocol in every possible detail and instead focused on providing enough information to understand the basics. I hope this analysis is useful to you and would love to hear your thoughts.

I did this analysis out of my private interest. My intention in writing this paper was to improve the trust in Threema. I invite you to judge about its safety on your own.

If you found a mistake or have any comments about this paper I'd be happy if you contact me via email[3]. Please contact Kasper Systems GmbH directly if you have found any (potential) weaknesses in their protocol.

[3] My E-mail address and PGP key are on the first page. Please don't contact me regarding the extraction of Threema keys and servers. As already mentioned they're not part of this paper and are sold by Kasper Systems GmbH.