CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Security Analysis of the Telegram IM
Student:	Bc. Tomáš Sušánka
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2017/18

Instructions

Study the state of modern Instant Messaging applications. Select a few suitable examples and document their main properties, with particular focus on the security of communication. To a greater depth study the Telegram IM application and its security protocol MTProto. Describe its structure and properties and compare the actual implementation with the official documentation. Evaluate your findings with respect to a possible application of attacks such as Padding Oracle, Timing Attacks or similar. If you find a weakness, attempt to demonstrate it practically using a suitable tool. Describe your discoveries and if necessary, propose a fix for found weaknesses.

References

Will be provided by the supervisor.

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 8, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Master's thesis

Security Analysis of the Telegram IM

Bc. Tomáš Sušánka

Supervisor: Ing. Josef Kokeš

4th January 2017

Acknowledgements

I would like to thank my thesis supervisor Ing. Josef Kokeš for his essential help, guidance and an incredible response time. Many thanks to Jakob Bjerre Jakobsen from the Aarhus University for sharing his source codes of the MTProto illustrations that I partially modified. Thanks to Prof. Dr.-Ing. Christof Paar and Dr.-Ing. Juraj Somorovsky from the Ruhr-University Bochum for encouraging me to pursue this topic. Special thanks to Miriam for proofreading and last but not least, I would like to thank my family for supporting me during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Žižkov, Prague on 4th January 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Sušánka, Tomáš. *Security Analysis of the Telegram IM.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Tato diplomová práce se zabývá studiem programu Telegram a s ním souvisejícího protokolu MTProto. Zaměřuje se především na kryptografické zázemí protokolu, zdrojový kód Android aplikace, zkoumá datový přenos a porovnává stav aplikace s oficiální dokumentací. Dále analyzuje potencionální bezpečnostní slabiny a případně demonstruje jejich zneužití.

Klíčová slova instant messaging, Telegram, MTProto, bezpečná komunikace, bezpečnostní analýza

Abstract

This thesis is devoted to an analysis of the Telegram Messenger and the related MTProto protocol. It studies the cryptographic background of MTProto, the Android client source code and the generated network traffic. Additionally, it compares the application to its official documentation. Finally it discusses potential vulnerabilities and various attempts to exploit them.

Keywords instant messaging, Telegram, MTProto, secure communication, security analysis

Contents

List of Figures

List of Tables

Introduction

Instant messaging is a form of a real-time online communication. The history of Instant Messengers (IM) dates back as far as 1960s, but the modern IMs as we currently know them started in the mid-1990s with the very popular ICQ or AOL Instant Messenger. Nowadays, the human interaction and communication take place more and more often in the digital realm and instant messaging is the digital version of a spoken dialogue.

A common feature of the IM software is the ability to see if a specific contact is online and available for chat. Further, IMs are primarily designed for an alternating exchange of a number of shorter messages, as opposed to the well-established and typically longer emails.

Today, many social networks provide some form of messaging solution as well. The users are very demanding. Instant messengers are de facto required to have both mobile and web access and more advanced capabilities, such as media, voice, or location sharing. Furthermore, the exposure of mass surveillance leaked by the ex-NSA employee Edward Snowden fuelled the demand for a secure and privacy-aware communication.

This thesis aims to provide a brief overview of the current Instant Messenger solutions, focusing on their security. We will cover the messengers' history, origin, their security-related aspects and other information in order to provide the reader with an insight into the security of today's modern instant messaging applications.

Following this summary we will focus in more depth on Telegram, which stresses the need for users' privacy. Telegram has 100 million active users and is especially popular in Iran, but also in the US, Germany, India, Uzbekistan, Russia, Italy or Brazil [1]. Telegram introduces its own cryptographic protocol MTProto instead of using already known solutions; this was criticised by a significant part of the cryptographic community [2, 3]. We will devote ourselves to its in-depth investigation, study of the cryptography behind the protocol, how the protocol deals with a key negotiation, message authentication, integrity and other topics.

Furthermore, we will dive into the code of Telegram itself, focusing primarily on the official Android application. We will verify its consistency with the official documentation, focus on its code base, evaluate how it proceeds with storing user's data, and finally we will draft an attack scenario and attempt to execute it.

Chapter 1

Current security status of major IMs

This chapter contains a thorough description of five selected Instant Messengers and their security related discoveries. For an easier comparison we decided to select messengers with support for mobile platforms. Individual software versions mentioned are mostly an estimate based on a date some findings were published and on the software changelog.

1.1 Selection

For our evaluation we selected five Instant Messenger applications. The selection was based on various criteria to create a diverse mixture of messengers. These criteria were amongst others: user base, geographical origin, authors, proclaimed security, license and price. However, we did not want to simply sort the IMs according to either one of those criteria. We wanted to present a diverse collection. That is why we omitted the Facebook Messenger and Apple's iMessage, since the first is owned by the very same company as WhatsApp [4] and the second is yet ano ther US-based company.

1.1.1 WhatsApp[1]

A large user base and overall popularity of the application is one of the main reasons WhatsApp is included. With 1 billion active users it is the most popular messenger at the moment [5].

[1] https://www.whatsapp.com

1.1.2 Signal[2]

Signal was endorsed by the community on several occasions and is considered the most secure messenger by many [6]. It is the only completely open-source messenger in this collection.

1.1.3 Threema[3]

Threema is the only paid application in this selection. Furthermore, Threema comes from Switzerland, and therefore may be considered as an European alternative to the traditionally US-based services.

1.1.4 WeChat[4]

WeChat is a China-based messenger and with its 700 million active users the most popular there [7]. As with Threema, we include WeChat mainly for its distinct origin.

1.1.5 Telegram[5]

Telegram developers proclaim it is safer than WhatsApp. It uses its own messaging protocol MTProto and argues for its security. Telegram's clients are open-source but the server side is proprietary. The creators of Telegram are Nikolaj and Pavel Durov, the authors of the Russian social network VK.

Table 1.1: Messengers

Name	**First release**	**License**	**User base**
WhatsApp	January 2010	Proprietary	1 billion[6]
Telegram	August 2013	GPLv2/GPLv3/Proprietary	100 million[6]
Signal	July 2014	GPLv3	10 million[7]
Threema	December 2012	Proprietary	3.5 million [8]
WeChat	January 2011	Proprietary	700 million[9]

[2]https://www.whispersystems.org
[3]https://www.threema.ch
[4]https://www.wechat.com/en/
[5]https://www.telegram.org
[6]As of February 2016.
[7]Signal's predecessor TextSecure as of December 2013.
[8]As of June 2015.
[9]As of April 2016.

1.2 Security aspects

The Electronic Frontier Foundation maintains a scoreboard of messaging applications' security. The scoreboard was initially released on November 2014 and last updated on 5th June 2016.

It evaluates messengers based on these seven criteria [8]:

- **Are messages encrypted in transit?** All user communication is required to be encrypted. Encryption of metadata, such as phone numbers, usernames or dates, is not required.
- **Are messages encrypted in such a way the provider cannot access it?** All user messages need to be end-to-end encrypted from the moment user sends a message to the moment the other party receives it. No decrypting and re-encrypting may occur during that process. The private keys need to be generated at the endpoints, not at a centralized server. Bulk data collection is therefore meaningless and no third-party can access the messages unless one party allows it.
- **Can user verify contacts' identities?** This requires a verification mechanism of the opposite side's identity to prevent Man-in-the-Middle attacks.
- **If the keys are stolen, are past communications secure?** All messages need to be encrypted with routinely changing keys. The forward secrecy minimizes the consequences in a case a private key is stolen because each key has only a short-time validity. This criterion requires end-to-end encryption and is therefore directly dependent on the second criterion.
- **Is the code open to an independent review?** A sufficient amount of source-code needs to be available to perform an independent code review. This provides protection from unintentional encryption flaws, backdoors or bugs.
- **Is the cryptography design properly documented?** The cryptography behind the application needs to be placed on record in detailed documentation.
- **Has there been any recent code audit?** An independent security review of the application must not be older than 12 months. It is not required that the audit is publicly available.

At the end of each chapter dedicated to one messenger a small note about the received score will be made.

1.3 WhatsApp

WhatsApp is a mobile messaging application. Besides text it enables users to send pictures, videos, voice and locations. WhatsApp Messenger is available for iPhone, BlackBerry, Windows Phone, Android and Symbian [9].

In February 2016, WhatsApp has reached 1 billion active users per month and was the most used messenger to date [5]. Its large user base and overall popularity of the application is one of the main reasons WhatsApp was included in this comparison.

Following section describes WhatsApp's security-related incidents.

1.3.1 Security-related incidents

1.3.1.1 Flaws in the registration process

WhatsApp's user identity is bound to the user's phone number. In order to verify the connection every user has to enter their phone number during the first start-up. WhatsApp server then sends SMS with a verification code to this number. User submits the code from the received text message and WhatsApp creates their account.

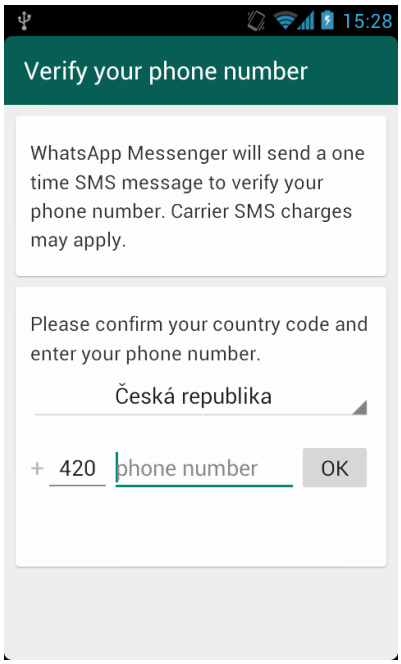


Figure 1.1: WhatsApp's registration process prompts the user to enter his phone number, and verifies it by sending a text message.

Up to version 2.6.5, WhatsApp offered an alternative verification process. Instead of the user waiting for a text message, he was supposed to send an SMS

to one of WhatsApp's phone numbers where he included his email. WhatsApp later sent verification code to the email and user verified himself with this code [10].

Serious issues were found with this method in 2011 [11]. To hijack user's account an attacker chose this verification method. Then using an SMS spoofing service he sent an SMS to a WhatsApp's phone number pretending it originated from the victim. The attacker then entered his own email in the message, resulting in WhatsApp sending the verification code to the attacker. The attacker then simply entered the code from the email and successfully hijacked the victim's account.

Following these findings another method for bypassing the registration process was revealed. During the registration phase WhatsApp sent a request for a verification SMS to be sent to the client in HTTP request similiar to this [10]:

```
GET [..]?to=4915143[..]&auth=659&[..] HTTP/1.1
User-Agent: WhatsApp/2.6.4 iPhone_OS/4.3.3 Device/iPhone_4
```

Listing 1: HTTP request to dispatch a text message to a client for verification.

The request contained a final verification code in the GET parameter. This led to a conclusion the client created the verification code, not the server, and expected user's confirmation. An attacker could simply intercept the request, retrieve the verification code and make sure the request did not arrive to WhatsApp's servers to make the victim unaware of his malicious intentions.

The attacker created a fake HTTP OK response to let the messenger think the request was successful and then entered the retrieved verification code from the intercepted request. The attacker successfully hijacked the victim's WhatsApp identity and could both send and receive all the victim's messages.

The author of the attack scenario notified WhatsApp developers beforehand and WhatsApp fixed the issue before the research was made public [10]. At the time of writing this thesis, WhatsApp did not offer the discussed verification method.

1.3.1.2 Password generation

WhatsApp uses a slightly modified version of the XMPP protocol [12]. During the registration process it creates a username based on the user's phone number. In versions newer than 2.10 the password is generated server-side [13], older versions however, used the phone's IMEI number as a password [13, 14]. Any phone number and IMEI was therefore everything an attacker needed to send messages on victim's behalf. Numerous applications are collecting plenty of user data, and the IMEI and phone number might be amongst them. Any

database leak of such information would lead directly to a mass accounts abuse.

```
md5(revert(<IMEI>))
```

Listing 2: Pseudo-code of the password generation on Android in older versions of WhatsApp.

1.3.1.3 Messages encryption

Up to approximately version 2.8, WhatsApp did not use any message encryption. The messenger used port 443 (commonly used for HTTPS) to send content, but it did not encrypt anything at all [15]. Using a simple network sniffer like Wireshark an attacker was able to read all of the user's messages.

In May 2012, an application called WhatsAppSniffer was released [16, 17]. It exploited the previously described flaw and enabled the attacker to see all of the victim's messages in an easy and comprehensible user interface.

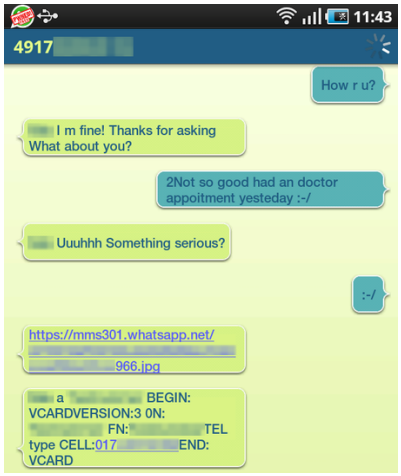


Figure 1.2: WhatsAppSniffer application was able to sniff all the user's data and read them [17].

In August 2012, WhatsApp first started to use some sort of encryption. The developers did not reveal which protocol they used or any other information about it. Reports showed that simple message sniffing, as described in the previous paragraph, ceased to work [18]. Some sources claim the RC4 stream cipher was used for encryption [19, 20].

On November 18th, 2014, the creators behind Signal messenger, Open Whisper Systems (OWS), announced a partnership with WhatsApp. The partnership should have resulted in incorporating OWS' encryption protocol

into WhatsApp, bringing the end-to-end encryption to all WhatsApp clients. Open Whisper Systems stated: *"We are moving quickly towards a world where all WhatsApp users will get end-to-end encryption by default"* [21]. At the time WhatsApp confirmed the partnership, but did not comment on it any further or offered any further information. WhatsApp's FAQ only briefly stated that *"WhatsApp communication between your phone and our server is encrypted"* [22].

In April 2015, *heise.de* investigated the current state of WhatsApp's encryption. The journalists were sniffing messages using the Man-in-the-Middle technique. They showed that Android versions used end-to-end encryption and that the messages were encrypted according to the TextSecure protocol [23]. However, during an analysis of the iOS client they concluded the messages weren't protected in such a manner. Finally, they concluded that they were unsure whether end-to-end encryption was actually used in all cases.

In April 2016, WhatsApp finally released an official white paper confirming all its messages are end-to-end encrypted with the use of Signal Protocol. The document thoroughly described all aspects and encryption methods. It stated *"WhatsApp messages, voice and video calls between a sender and receiver that use WhatsApp client software released after March 31, 2016 are end-to-end encrypted."* [24].

1.3.2 EFF's secure messaging score

At the time of writing, WhatsApp has six points out of seven in the EFF's secure messaging scorecard [8].

Table 1.2: WhatsApp's secure messaging score

Are messages encrypted in transit?	✓
Are messages encrypted so the provider can not access it?	✓
Can user verify contacts' identities?	✓
Are past communications secure if keys stolen?	✓
Is the code open to independent review?	✗
Is the cryptography design properly documented?	✓
Has there been any recent code audit?	✓

1.4 Signal

Signal is an open-source voice calling and messaging application. It is available for both Android and iOS. Its messages are end-to-end encrypted. During a voice call a simple identity check is available when a given word is to be pronounced by both sides of the call.

Signal uses its own Signal Protocol (previously called Axolotl) based on cryptographic primitives such as Elliptic Curves (Curve25519), AES and HMAC-SHA256 [25]. As mentioned in Section 1.3, WhatsApp currently uses the very same protocol.

1.4.1 Security-related incidents

In late 2014, *Der Spiegel* published several articles claiming NSA considered Signal's encrypted voice calling combined with other tools as a serious hindrance to theirs surveilance missions [26]. Edward Snowden recommended Signal on various occasions [27, 28].

A research team from the Ruhr-University Bochum provided a security analysis of the former Signal version titled *"How Secure is TextSecure?"* [25]. They came to a conclusion that the protocol is susceptible to an Unknown key-share attack and proposed a correction. The researchers pronounced Signal as secure in case the suggested patch was implemented, which it eventually was.

Up to this date no other security-related incidents or further analysis of Signal messenger are known.

1.4.2 EFF's secure messaging score

Signal fulfilled all the requirments in the EFF's secure messaging scorecard and received a full score [8].

Table 1.3: Signal's secure messaging score

Are messages encrypted in transit?	✓
Are messages encrypted so the provider can not access it?	✓
Can user verify contacts' identities?	✓
Are past communications secure if keys stolen?	✓
Is the code open to independent review?	✓
Is the cryptography design properly documented?	✓
Has there been any recent code audit?	✓

1.5 Threema

Threema is a paid IM application available for three major platforms – Android, iOS and Windows Phone. Text messaging, multimedia, locations, voice messages and file sharing are all supported. Threema is native to Switzerland and all servers are located there.

Threema is a paid application. As of December 2016, the price was set to a single payment of €2.99. Threema is using user IDs and linking user's phone number and email address is not required.

1.5.1 Security-related incidents

In August 2015, Threema was audited by an independent company. The Threema documentation states [29]: *"The result confirms that Threema's concepts fully meet the requirements for secure and trustworthy instant messaging."* The full audit is available on Threema's web page.

The main point of Threema's criticism is considered to be its closed-source nature and subsequently the inability to verify its source code. There are no other security analyses.

1.5.2 EFF's secure messaging score

In the EFF's comparison Threema missed a single point for not completing an indenpendent code review [8].

Table 1.4: Threema's secure messaging score

Are messages encrypted in transit?	✓
Are messages encrypted so the provider can not access it?	✓
Can user verify contacts' identities?	✓
Are past communications secure if keys stolen?	✓
Is the code open to independent review?	✗
Is the cryptography design properly documented?	✓
Has there been any recent code audit?	✓

1.6 WeChat

WeChat is a Chinese mobile messenger available to all major mobile platforms including iOS, Android, Windows Phone and BlackBerry. Besides regular text messages, WeChat offers media, location, video sharing and even games. As of May 2016, WeChat had 700 million active users [7].

A special business versions of WeChat named Enterprise WeChat was introduced in April 2016, designed for communication within companies and offering few additional features.

1.6.1 Censorship

The Internet censorship in China blocks extensive amount of web sites including web giants such as Facebook, Twitter or Google [30, 31].

Since WeChat's servers are located in China, it is subjected to these restrictions. Users are required to agree to and obey specific rules, such as *"upholding the socialist system, social morality and authenticity of information"* [32]. The Chinese government's reasoning for such actions is allegedly to build a cleaner cyberspace and to ensure national security [33].

1.6.2 Security-related incidents

WeChat operators are capable of accessing messages, contacts and even user's location. Numerous countries including United States, India and even China itself expressed concerns over WeChat being a threat to their national security issues [34].

1.6.2.1 XcodeGhost malware

In 2015, Apple reported WeChat was infected with XcodeGhost malware [35]. The malware was able to obtain user device information, read clipboard, prompt alert dialog and other. Apple claimed the malware was not able to cause any significant damage and could not access any user data.

1.6.3 EFF's secure messaging score

Regrettably, WeChat was not included in the EFF secure messaging survey.

1.7 Telegram

Telegram is an instant messaging service enabling users to send messages, photos, videos, stickers and files. Telegram describes itself as fast and secure solution for instant messaging and claims to be safer than WhatsApp. Compared to WhatsApp, Telegram is more cloud-based because it stores all messages on its servers and synchronizes them with all of the user's devices [36].

Nikolaj and Pavel Durov are the authors of Telegram. After leaving the social network VK founded by Pavel Durov, they focused on creating safe forms of communication, eventually resulting in Telegram.

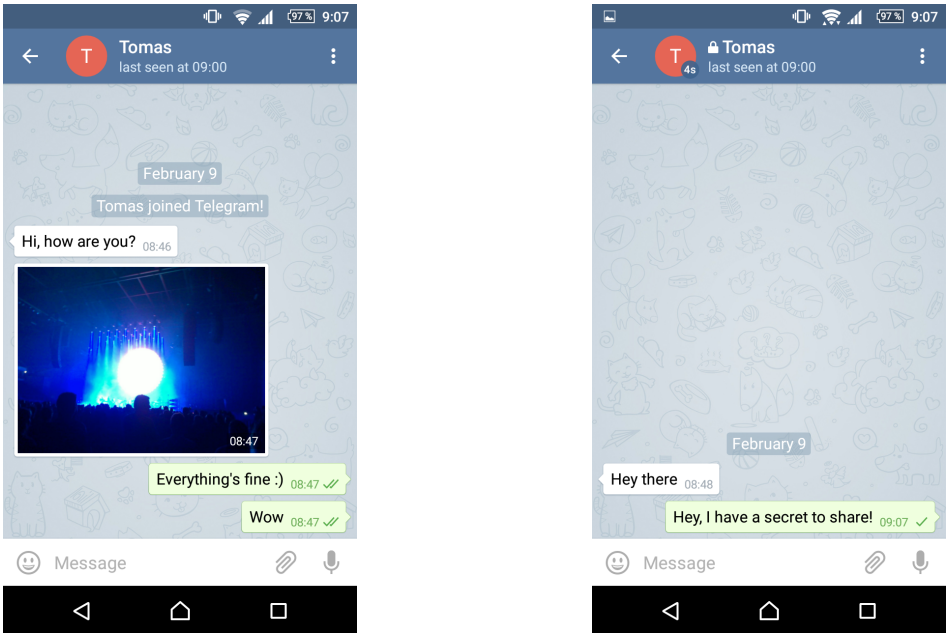
Telegram provides two modes of messaging. Apart from the regular chat, Telegram provides so-called *secret chats*. Secret chat messages are end-to-end encrypted and are not stored on the Telegram's servers for longer periods of time than absolutely necessary [36].

Similar to WhatsApp, user can contact someone using his phone number, but Telegram provides classical username-focused approach as well. User needs to know the recipient's phone number or Telegram username in order to communicate with them.

All clients are licensed under GPLv2 or GPLv3 license, the server-side part of Telegram is a closed-source and proprietary software [37].

In 2015, a Brazilian judiciary commanded WhatsApp to shut down its services for 48 hours. During this period, which was ultimately lowered to only 12 hours, Telegram welcomed 5 million new users [38]. It may therefore be considered a direct competitor of WhatsApp.

In May 2015, Telegram had 62 million active users [39].



(a) Regular chat

(b) Secret chat

Figure 1.3: Telegram has two chat modes. Regular chat is meant to be used for traditional communication and its messages are stored on Telegram's servers. Secret chats are supposed to provide another layer of protection and are end-to-end encrypted.

1.7.1 Security-related incidents

1.7.1.1 SMS authentication

In Section 1.3.1.1 we described how the WhatsApp's registration process using SMS works. Telegram works on a similar basis and allows logging in using a simple SMS as well.

The text messages are fully accessible for mobile network operators who can and usually do cooperate with the corresponding government. Furthermore, SMS can be intercepted using inexpensive IMSI catchers. If an attacker is able to steal the authenticating SMS, he is able to login on the user's behalf. What is worse, Telegram stores the whole of user's message history, thus it allows the attacker to read older messages as well.

In early 2016, it was shown to be a legitimate concern [40]. Some Telegram users experienced an erasure of their accounts and blamed Telegram of enforcing political censorship which Telegram denied. Russian activist Oleg Kozlovsky described in his Facebook post [41] how his account was allegedly hacked.

First, the Russia's mobile operator disabled SMS service on Oleg's phone number. Upon the disconnection the attacker tried to log to the victim's

Telegram account. Because intercepting the SMS is all the attacker needs he simply entered the authorization code he sniffed and got full access to Oleg's account. Telegram recommended to turn on two-factor authentication for additional security.

These concerns raise a serious question whether SMS authentication is a valid instrument security-wise.

1.7.1.2 Cracking Contest

On November 4th, 2014, Telegram arranged a contest with a winning price of $300,000 for cracking its encryption. The contest became quite known in the community, and probably provided a bit of an advertisement for Telegram.

The contest remained unsolved until its closure. Number of authors considered it rigged and stated that the contest does not provide any proof of Telegram's overall security whatsoever [42, 43].

1.7.1.3 IND-CCA insecurity

In Spring 2015, researchers from Aarhus University performed an independent audit of the protocol [44]. They concluded that the encryption scheme is not IND-CCA[10] secure, meaning any ciphertext can be altered into another ciphertext decrypting to the very same plaintext.

The researchers stressed the theoretical nature of the attack and that they *"do not see any way of turning the attack into a full plaintext-recovery attack"* [44]. Telegram's FAQ describes it as a minor issue unaffecting the overall security [45].

1.7.2 EFF's secure messaging score

As mentioned Telegram has two types of messages. Telegram is therefore evaluated twice by the EFF.

Table 1.5: Telegram regular chat's secure messaging score

Are messages encrypted in transit?	✓
Are messages encrypted so the provider can not access it?	✗
Can user verify contacts' identities?	✗
Are past communications secure if keys stolen?	✗
Is the code open to independent review?	✗
Is the cryptography design properly documented?	✓
Has there been any recent code audit?	✓

[10] Indistinguishability under Chosen Ciphertext

Table 1.6: Telegram secret chat's secure messaging score

Are messages encrypted in transit?	✓
Are messages encrypted so the provider can not access it?	✓
Can user verify contacts' identities?	✓
Are past communications secure if keys stolen?	✓
Is the code open to independent review?	✓
Is the cryptography design properly documented?	✓
Has there been any recent code audit?	✓

Chapter 2

Cryptography behind Telegram

Telegram authors decided to craft a brand new encryption scheme, MTProto, in order to supposedly achieve better delivery times and stability. In this section we will describe in detail how the protocol should work based on the official Telegram documentation and on the research from Aarhus University [44].

2.1 Initialization

During the first launch of Telegram application user needs to enter and verify their telephone number. The verification is done by sending a five digit code to the phone via SMS. The user then enters the code into the app and verifies the phone number. When this process is done, the registration process begins as follows:

Suppose a client C is registering to a server S:

1. C sends a 128-bit random integer *nonce* to S.
2. S responds with another 128-bit random integer *server_nonce*, composite number pq and a fingerprint of a RSA public key.
3. C provides a proof of work by decomposing pq into prime factors p and q such that $p < q$.
4. C has several RSA public keys stored locally, and chooses the appropriate one based on the received fingerprint.
5. C crafts a payload of p, q, pq, *nonce*, *server_nonce* and another new 256-bit random integer *new_nonce*. The payload including its hash is then encrypted by the RSA key and sent to S.
6. S responds with Diffie-Hellman parameters g, p and g_a encrypted with AES-IGE using a key derived from *new_nonce* and *server_nonce*.

7. C generates a random 2048-bit b, and computes $g_b = g^b \bmod p$ and $K = g_a^b \bmod p$. The g_b value is sent to S encrypted as previously described.
8. S calculates $K = g_b^a \bmod p$. Both C and S now share a common key K called as `auth_key`.

Some details were omitted for simplification. During the exchange the client should check the following requirements:

- p is a safe prime, meaning $\frac{p-1}{2}$ needs to be prime as well
- $2^{2047} < p < 2^{2048}$
- g is equal to 2, 3, 4, 5, 6 or 7 and it generates a cyclic subgroup of prime order $\frac{p-1}{2}$
- $1 < g_a, g_b < p - 1$
- and finally, it is recommended to check $2^{2048-64} < g_a, g_b < p - 2^{2048-64}$

The requirements check might be cached, and the client is allowed to hard-code the already checked p and q values into the application as well.

We compared these requirements with the FIPS 186-4 publication concerning Digital Signature Algorithm and concluded it requires similar conditions as Telegram with only minor differences.

The resulting `auth_key` is now used for all client-server communication and regular chats. Finally, a fingerprint of the exchanged `auth_key` is created labeled as `auth_key_id`. It is crafted from the last 64 bits of `SHA1(auth_key)`.

2.2 Regular chats

With the connection to the server established in the previous section we can now look into the delivery process. The Aarhus research [44] was concerned with secret chats only, therefore in this section we rely solely on the official documentation.

2.2.1 Payload

First, let us describe the content of a single message further referred to as a *payload*. The payload prior to encryption is depicted in Figure 2.1 and it contains:

- **salt** Periodically changed number used for various protection purposes.
- **session_id** Unique number to identify the user and their device.
- **msg_id** Unique ID of the message within a session.

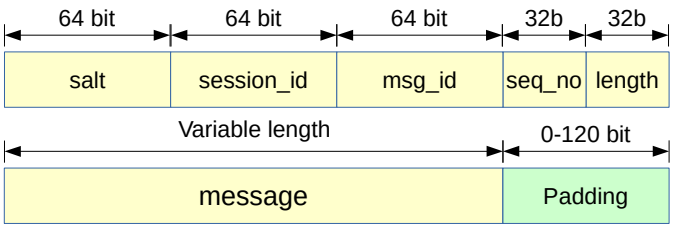


Figure 2.1: Payload of a single message to be encrypted in the regular chat contains salt, session and message identifier, sequence number, length and the text itself.

- **seq_no** Message sequence counter.
- **length** Length of the actual message.
- **message** Content of the actual message.

2.2.2 Encryption

The whole encryption process is visualized in Figure 2.2.

First, the message key `msg_key` is calculated. It is composed of the 128 least significant bits of the SHA-1 hash of the payload to be encrypted. Next, the array is padded with 0-120 random bits in order to be divisible by the AES block size – 128 bits.

This `msg_key` along with the `auth_key` described in Section 2.1 are taken as input for the Key Derivation Function (KDF) which performs a number of SHA-1 hashes and truncations, yielding two 256-bit values: the AES key and the IGE initialization vector (IV) used for encrypting this particular message.

Finally, two other values are added on top of the encrypted data array: the already mentioned `msg_key` and the `auth_key_id` described in Section 2.1. The data array is then ready to be transported to the server.

2.2.2.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric block cipher with key lengths of 128, 192 and 256 bits respectively, and a block size of 128 bits. Telegram uses 256 bit key. AES is currently the most widely used symmetric cipher and is used in many internet standards such as IPsec, TLS, SSH and number of others [46].

AES was selected through a worldwide open competition and finally defined in the FIPS-197 publication by the US National Institute of Standards and Technology (NIST) in 2001. As of today, AES is secure against brute-force attacks and no viable analytic attacks are currently known [46].

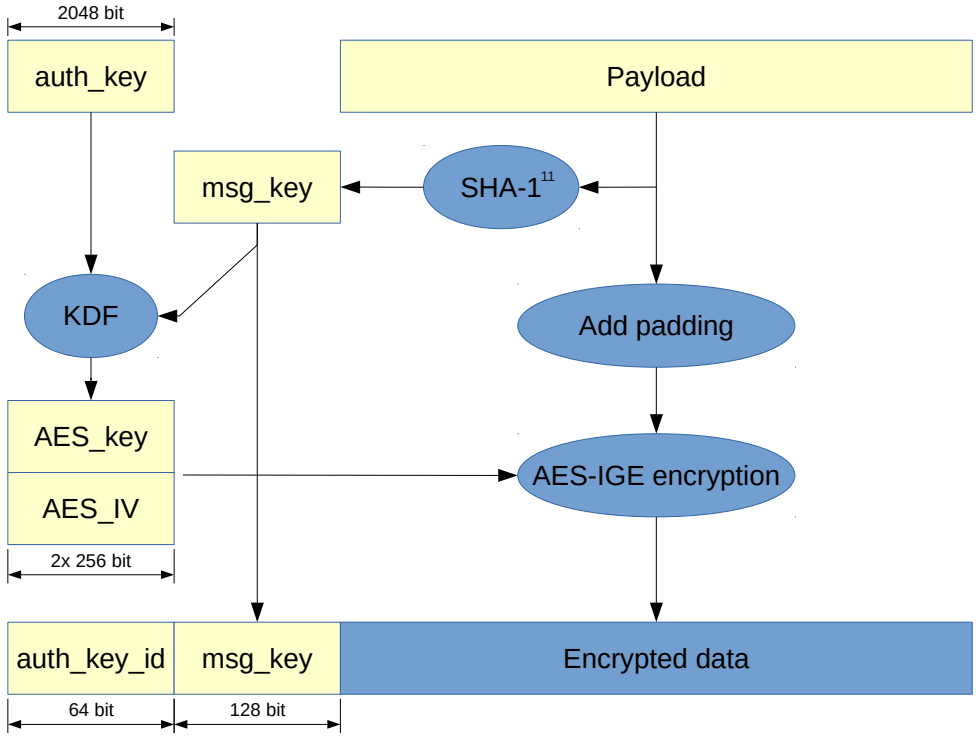


Figure 2.2: The AES key and IV is derived from the `auth_key` and SHA-1 hash of the payload `msg_key`. Using these values the padded payload is encrypted. The encrypted data along with `auth_key_id` and `msg_key` are ready to be transported [44].

2.2.2.2 Infinite Garble Extension

Infinite Garble Extension (IGE) is a lesser-known block cipher mode. It is shown on Figure 2.3 and defined by the following formula [47]:

$$c_i = f_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$$

where f_K stands for the encrypting function with key K (AES in our case), and i goes from 1 to n – the number of plaintext blocks.

A careful reader notices that for the first output block we need two initialisation values m_0 and c_0. Both are taken from the IV values described earlier. The original paper described m_0 as a random block and c_0 its encrypted counterpart. The OpenSSL implementation, however, uses a more general implementation where both m_0 and c_0 are provided by the user [47].

[11] The SHA-1 output is truncated to the first 128 bits.

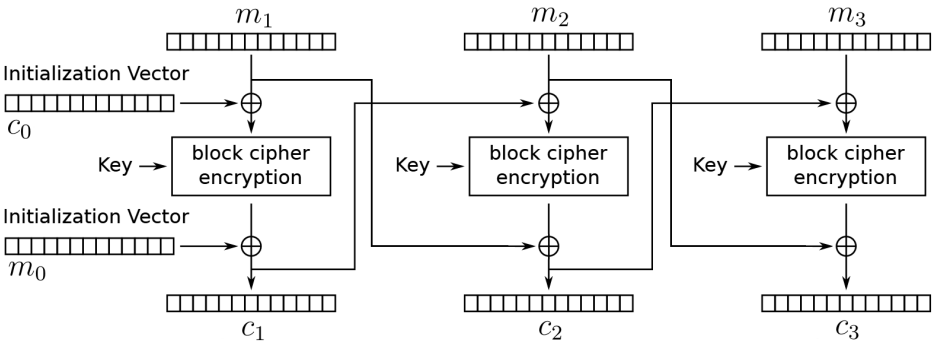


Figure 2.3: Infinite Garble Extension block mode with two Initialization Vectors [44].

2.2.2.3 SHA-1

SHA-1 is the most widely used cryptographic hash function [46]. It produces a 160-bit digest of a message. The authors opted for SHA-1 because of its low resource consumption [48]. Some experts consider SHA-1 as outdated and recommend its replacement by SHA-2 or SHA-3 [48].

2.2.3 Decryption

Before the decryption process starts the `auth_key_id` is validated. The receiver's `auth_key_id` needs to be matched against the value the sender appended to the byte array. If the values differ, the whole message is discarded.

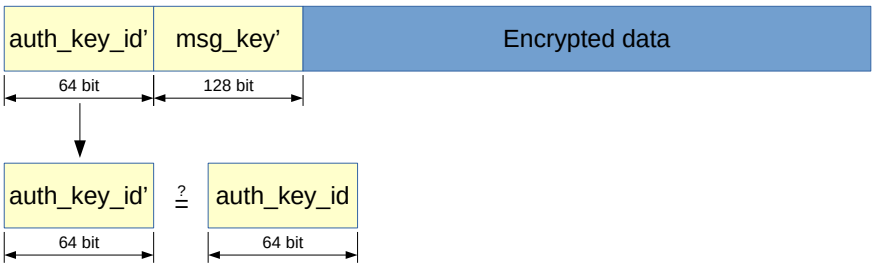


Figure 2.4: Before the decryption process starts the `auth_key_id` values are checked.

The decryption process is, to put it plainly, the encryption process in reverse. The KDF yields the AES key and IV values used for decrypting the data. The padding is stripped and the SHA-1 of the payload generates `msg_key` which is compared to `msg_key'`. The whole process is accurately visualized in Figure 2.5.

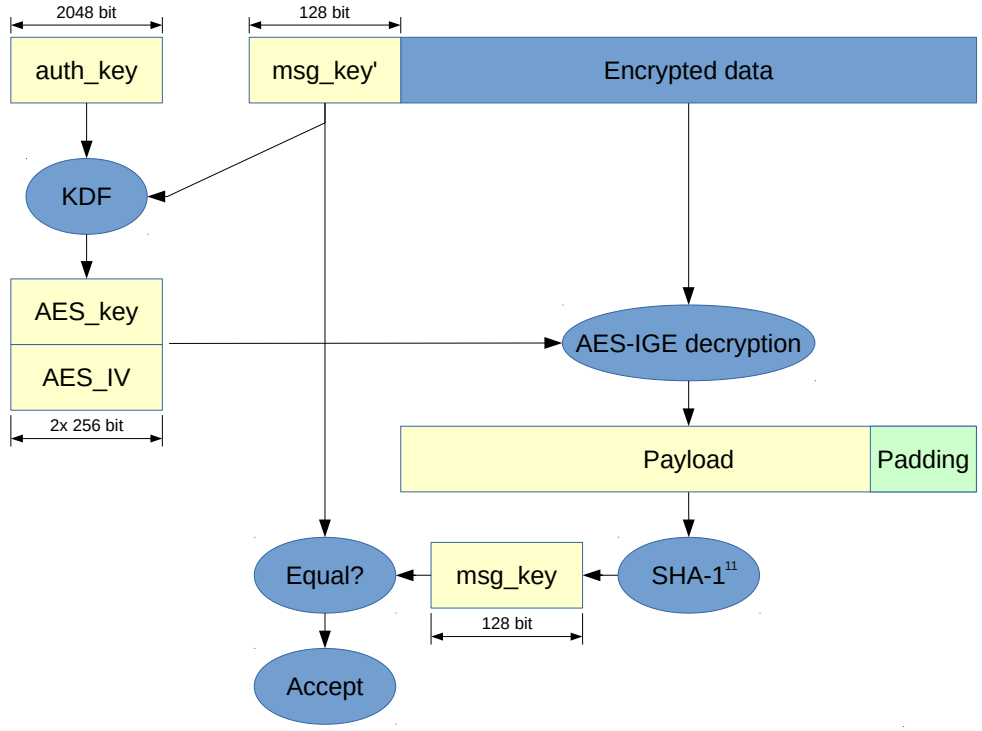


Figure 2.5: The KDF produces the AES key and IV from the `auth_key` and received `msg_key'`. The SHA-1 of the decrypted payload generates `msg_key` which is compared to `msg_key'` and only then accepted [44].

2.3 Secret chats

Secret chats are designed to bring an extra level of security compared to the regular chats. A secret chat can be initiated between two particular devices only, and therefore the messages can be read on those devices exclusively.

It is essential to note that all the secret chat communication is done using the established connection described previously in Section 2.1. All data are considered as an input into the MTProto protocol for regular chats, and therefore encrypted twice.

2.3.1 Key exchange

The exchange performs traditional Diffie-Hellman key exchange (DH). The DH parameters are received from the Telegram server and the client verifies them in the very same manner as in Section 2.1. The exchange proceeds similarly as earlier, however, with the connection already established is a little more straightforward. Suppose user A initiates a secret chat with user B:

1. A computes a random 2048-bit number a and sets $g_a = g^a \bmod p$.

2. B receives the request on all authorized devices and a single one accepts it.

3. B generates random b and sets $g_b = g^b \bmod p$.

4. Both users calculate the master key $K = g_a^b \bmod p = g_b^a \bmod p$. K is the secret chat's master key denoted `auth_key` as well.

What is worth mentioning, in secret chats key exchange the client generates its a (and b respectively) in the following way:

$$a = r_{client} \oplus r_{server}$$

Where r_{client} is a 2048-bit random integer generated on the client's side and r_{server} is 2048-bit random integer generated on the server's side. These values are then XORed, constructing the client's secret value. This is done to mitigate the client's poor capabilities to generate a cryptographically secure random number. This concerned the Android platform in August 2013 [49].

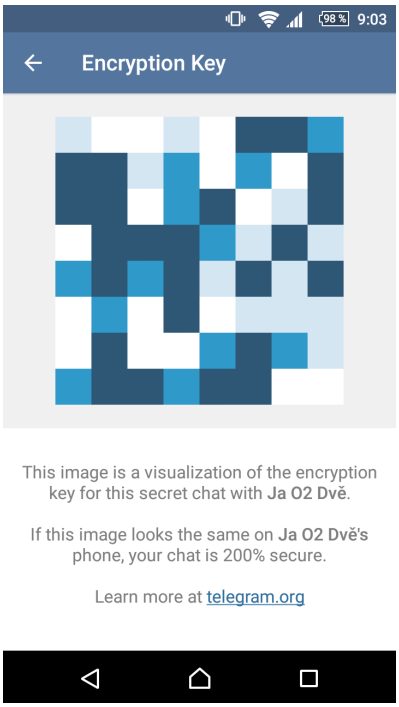


Figure 2.6: Encryption key visualization provides a technique to verify no malicious intermediary is present.

The DH by itself does not provide any authentication of the communicating parties and is therefore susceptible to an active Man-in-the-Middle attack. To mitigate this issue the user is provided with an option to display their

counterpart's encryption key. Telegram creates a white-blue box to visualize the key as may be seen in Figure 2.6. To make sure no malicious mediator is present users are supposed to meet in person and verify that the keys are identical. There is no such mechanism in regular chats.

The visualisation used to be based on a 128-bit fingerprint[12] of the secret key. An article published in January 2015 showed [50] that when the attacker forces (e.g. socially engineers) both sides to initiate the secret chat, a MitM attack is possible only with 2^{64} operations rather than 2^{128}, thanks to the Birthday paradox. The article claims the attack is possible for a well financed adversary and estimates the attack to cost tens of millions of USD. Telegram's FAQ addresses this issue and states the cost is around a trillion dollars to achieve a result in one month. Nevertheless, Telegram later decided to use additional 160 bits from the key – summing the fingerpint up to 288 bits [45]. This improvement should make this attack impossible.

The article reasons as well that in a real-world scenario the users do not actually meet to verify the key. Most of them probably ignore the verification altogether, some send the image via regular chat or using another – possibly insecure – channel.

To accomplish the Perfect Forward Secrecy model the key is renegotiated each time it has been used for more than 100 messages or has been used for more than one week. For the renegotiation the already established chat is used to send the g_a and g_b values. The server does not help with the randomness here, and the same DH parameters are used. The *auth_key* visualization does not change and is therefore based on the first negotiated key.

It is unclear whether Telegram applies the Trust on First Use mechanism[13].

2.3.2 Payload

With the key negotiated we may now focus on the message encryption itself. The payload slightly differs from its regular chat counterpart. It is composed of:

- **length** The length of the payload excluding padding and the length itself.
- **payload type** Header related to the protocol version and the message type.
- **random bits** Up to 120 random bits generated by the author followed by 8 bits to specify the length of it in bytes.

[12] Not to be confused with the 64-bit `auth_key_id` fingerprint.

[13] Trust on First Use is a security mechanism used by a client software which unconditionally trusts the other side's identity during the first use. If the key changes later on, the user is alerted.

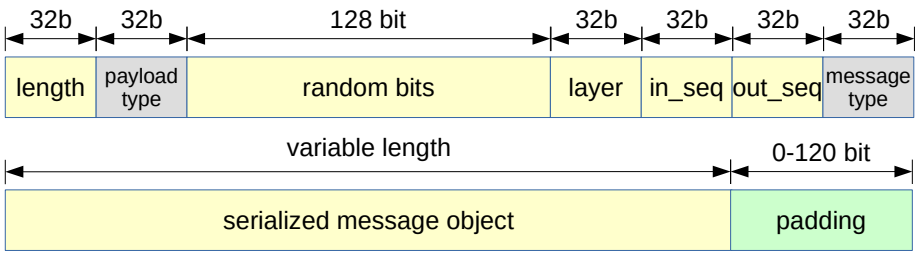


Figure 2.7: Payload of a single message in secret chat contains a number of additional fields [44].

- **layer** Integer specifying the protocol version.
- **in_seq** Message counter for incoming messages.
- **out_seq** Message counter for outgoing messages.
- **message type** Header related to the protocol version and the message type.
- **serialized message object** Containing other values and the message itself as depicted in Figure 2.8.

Both payload and message types are not described rigorously in the documentation. The Aarhus research [44] states they are both *"headers related to the version of the protocol"*. The serialized message object contains four values:

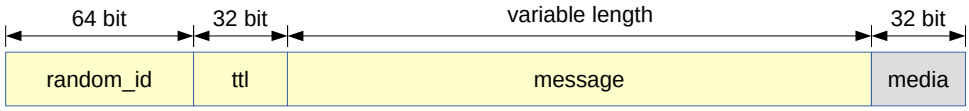


Figure 2.8: The message object contains the message itself, random integer, time to live settings and media header.

- **random_id** Random value assigned by the author to identify the message; also sent as a plaintext.
- **ttl** Time to live, an integer specifying the lifetime of a message in seconds. This concerns the so-called self-destruct mechanism.
- **message** The actual text of a message provided by the user.
- **media** Header specifying media attachment if any.

Throughout this paper we assume that no media attachments are present, and therefore the `media` attribute is always empty. This payload is then serialized as an array of bytes and prepared to be encrypted.

2.3.3 Encryption

The encryption process is the same as described in Figure 2.2. As mentioned at the beginning, the outcome of the encryption then serves as an input for MTProto regular communication.

2.3.4 Decryption

Decryption is the same as well. First the `auth_key_id`s are validated. Then the decryption process starts exactly in the same way as in Figure 2.4.

Chapter 3

The Analysis

In this chapter we present our analysis approach and share our findings. These findings are based primarily on these three sources:

- The official open-sourced Telegram application for Android
- The official Telegram documentation
- The Aarhus research [44]

We commenced by installing Telegram for Android and analyzed it with various tools in order to perform a comparison with the other sources mentioned above. The following section describes our actions.

3.1 Setup

First, we cloned the official Telegram repository. Because we realized a simple code exploration is not satisfactory due to both its complexity and a poor code quality, we installed the following software:

To run and potentially debug the application, the Android SDK is needed. Furthermore, the Android NDK is required as well since Telegram uses the Java Native Interface (JNI) which enables the program to incorporate low-level programming languages, such as C. For a complete developer-like experience the Android Studio was installed. Android Studio is the official IDE for Android platform and is based on the popular IntelliJ IDEA platform [51].

A mobile phone running a rooted Android operating system was connected to a computer with the mentioned software. The Android Studio itself deals with the phone-computer connection (using the *adb* software) and performs building, compiling and transferring the final .apk file to the phone. It enables the developer to debug the software as well which was crucial for better understanding of the Telegram's code.

Next, we wanted to sniff the data sent from the application. To achieve that two different approaches were used.

3.1.1 Sniffing using a wireless hotspot

In the first scenario we set up a wireless network on the computer, and instructed the cell phone to connect to it. Even though this does require the victim to connect to the network, there are ways to achieve this by using special dedicated devices [52]. The Diagram 3.1 shows this setup.

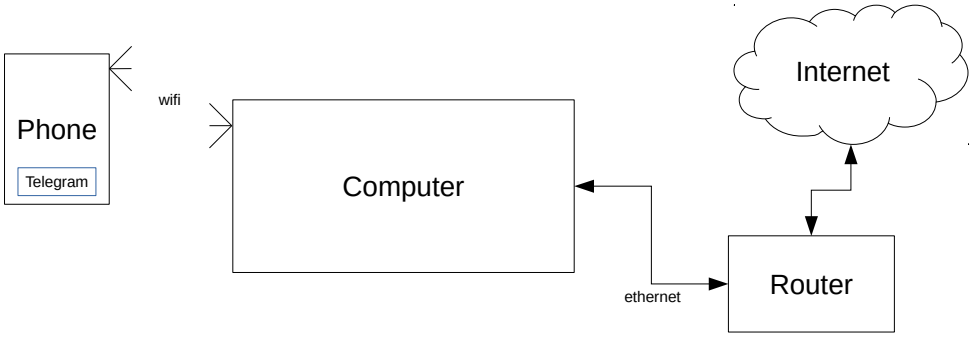


Figure 3.1: Computer running the analysis software is connected to the internet on one interface and creates a WiFi hotspot using another one. Cell phone running Telegram connects to the computer's WiFi, and all data are therefore routed through the computer.

This scenario is easy to set up because most operating systems support this functionality out of the box. The only downside is that we need two network interfaces – one for creating the hotspot and another one for internet connection.

3.1.2 Sniffing using ARP Poisoning

To demonstrate that we can omit the hotspot setup and perform the analysis without forcing the cell phone to change its network we are introducing a second scenario shown in Figure 3.2. In this case the computer and the mobile phone are in the same network.

By using Ettercap, a *"comprehensive suite for man in the middle attacks"* [53] available in Kali Linux, we launched an ARP Poisoning attack against the phone.

The Address Resolution Protocol (ARP) is a protocol which transforms an Internet layer address into a link layer address. Typically, this is the mapping of an IPv4 address to an Ethernet address. ARP Poising, or ARP spoofing, is a technique by which the attacker sends fake ARP packets to the victim, claiming his Ethernet address is associated with an IP address different than the one that is truly his.

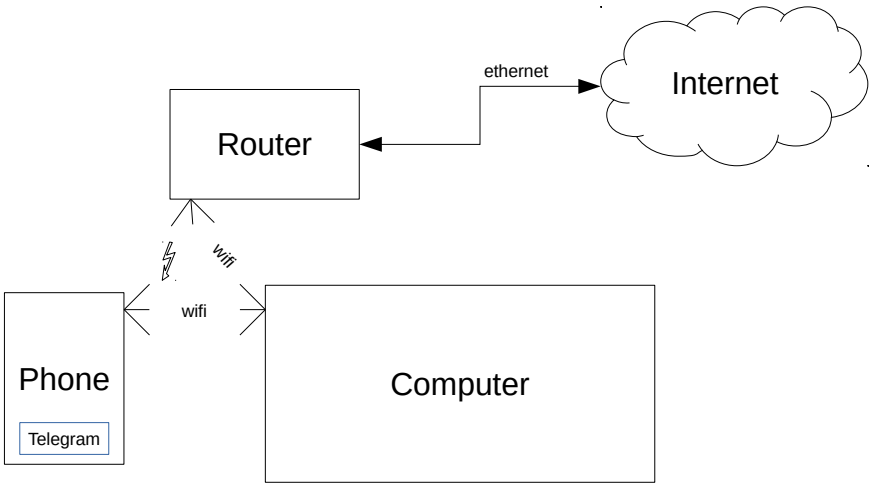


Figure 3.2: Both the computer and the cell phone are connected to the same network. Computer runs ARP Poisoning software inducing the phone to send traffic through the computer.

We launched this attack against the phone and the router, meaning the phone sent all of the communication to the computer instead of directly sending it to the gateway and vice versa. Since all communication is now flowing through the computer we may thoroughly analyze it.

This setup is also convenient if we are not able to use a WiFi as a hotspot, for example due to its necessity of it for the internet connection itself. Be aware that this setup should be used in test networks only.

3.2 Code

The application is publicly available on Github [54] with the first commit dating back to late October 2013. This thesis is concerned with the latest Telegram version 3.13.1 as of October 2016, the `64e8ec3` commit in particular.

The application's main programming language is Java but it uses the already mentioned JNI to incorporate C libraries such as boringssl, ffmpeg, sqlite and others. Since version 3.2.2, Telegram moved a significant portion of the logic into a separate C library called *tgnet*. The library is responsible for many connection-related actions including communication with the server and sending of messages. The codebase closely resembles the previous Java code. The secret messages are still dealt with within the Java part of the application.

It should be noted that this was introduced in the 3.2.2 version (September 2015) whereas the Aarhus analysis [44] is based on the 2.7.0 (April 2015) and therefore was not concerned with the *tgnet* library at all.

3.2.1 Code quality

We decided to analyze the code quality using static code analysis tools to avoid opinion-based reviews. Android Studio itself comes with a code scanning tool *lint* [55]. The tool scans for potential bugs, optimization improvements, security, performance and other.

Lint was run using the default settings and generated 20 errors and 286 warnings. The full output is available on the attached CD. Android Studio can perform code inspections as well. It runs Lint and adds a number of checks, such as error handling, security, code style, javadoc coverage, spelling etc. It found over 650,000 issues in the Telegram project. The results are available on the CD.

It is important to note that the code analysis concerns the project as a whole, meaning it includes all the dependencies. Some issues may therefore relate to the libraries Telegram uses, not to Telegram itself.

Furthermore, some files contain thousands of lines and multiple classes in one file. For example, the *TLRPC.java* has 23,010 lines containing 873 static classes. For a brief comparison we are including Table 3.1 comparing some of the Telegram metrics with Signal. The comparison excludes all libraries, binaries, assets etc. and discusses the source code related files only. We leave it to the reader to draw their own conclusions.

Table 3.1: Telegram vs Signal code metrics

	Telegram	**Signal**
Largest file by size	714.6 KB[14]	87.4 KB[15]
Largest file by line-count	23,009	2,096
Number of test files[16]	0	13

It should be noted that Signal's files are well documented which affects their size. That is not the case with Telegram at all.

3.3 Storage

In this section we briefly look at how Telegram stores its data. All data are stored in the phone's `org.telegram.messenger/files` folder. Two types of files are present – SQLite database files and `.dat` files which are in a Telegram's own configuration format:

- `cache4.db` SQLite database file.
- `tgnet.dat` Main configuration data.

[14] `TMessagesProj/src/main/java/org/telegram/tgnet/TLRPC.java`
[15] `src/org/thoughtcrime/securesms/util/Base64.java`
[16] Number of files each including at least one unit test.

- `dcXconf.db` Number of Datacenter specific configuration files.

The `cache4.db` file is a SQLite database file. It contains 38 tables in total, containing contacts, messages, chat information and all other user data. Messages are stored in bytes and therefore de facto in plaintext. Since the database is available only by physical access to the phone, we do not see this as a malpractice.

The `tgnet.dat` file is a place where Telegram stores its configuration directives. It contains configuration version, currently used datacenter ID, session IDs, time synchronization values, timestamps and other. For each datacenter – Telegram stored 5 during our analysis – it stores its ID, IPv4 and IPv6 addresses, salts and most importantly the secret `auth_key` and `auth_key_id` fingerprint.

3.3.1 Extraction scripts

To properly analyze the data Telegram stores we wrote three scripts, all in Python 3. The first script addresses the `tgnet.dat` file, the second the regular messages and the third the encrypted chat information, the last two work with the SQLite database. Examples of those files are included in the `examples` folder of the project and a brief readme file describing how to run the scripts is present as well. Some values' meaning is unclear, therefore the scripts print a simple question mark in such cases to indicate this.

3.3.1.1 Extracting tgnet.dat

The `tgnet-extractor.py` parses the `tgnet.dat` file and prints all the values to the standard output. This file has Telegram's own binary format and is, in a nutshell, a collection of values simply written one by one with no keys, formatting or structure. The most useful values for this research were the datacenter's IP addresses and the `auth_key` secrets, mainly the `auth_key_id` we expected to see in the sniffed traffic; more on that in Section 3.4.

3.3.1.2 Extracting messages

The script `message-extractor.py` works with the bytes stored in the SQLite database, the column *data* in table *messages* in particular. Those bytes store information on one single message sent in a regular chat.

The script extracts values, such as a message ID, values that identify both the sender and the receiver, timestamps, flags and most importantly the UTF-8 encoded string of the message itself.

3.3.1.3 Extracting secret chat information

Last but not least, the `encrypted-chat-extractor.py` script is responsible for the bytes in the column *data* of the *enc_chats* table. Amongst other things

it describes a secret chat instance and contains ID, dates, identification values and yet again an `auth_key` secret. Notice that this `auth_key` is used for this particular secret chat which is then encrypted with the regular chat `auth_key` as described in Section 2.3.

3.4 Undocumented obfuscation

During the data collection we found that the received data did not correspond with the documentation. We expected the data to be in a form of `auth_key_id`, `msg_key` and encrypted data as per Figure 3.3.

Figure 3.3: Expected form of sniffed data where 64-bits of key fingerprint are followed by `msg_key` and encrypted data.

Using the scripts described in 3.3.1 we extracted our testing `auth_key` and `auth_key_id` from our device in order to compare it with the sniffed data – the `auth_key_id` value was not present. The whole packet seemed random and therefore likely encrypted.

We examined the code and discovered the function responsible for this behavior is the `Connection::sendData` function located in `TMessagesProj/jni/tgnet/Connection.cpp` file. Before sending the data in the expected manner, they are encrypted one more time with a random key attached in front of the data. Interestingly, the Counter block mode is used instead of the IGE endorsed by Telegram.

To properly explain the obfuscation method we are introducing our own terminology first to add a little more clarity because Telegram does not practise a great job in naming variables. The reason is unclear but it may be done intentionally to make the deobfuscation process even harder.

3.4.1 Terminology

Since the variables are explained in more detail in the following sections, reader may skip this list and use it as a reference later on. The titles we are introducing are:

- **obf_enc_key_bytes** 64 random bytes (512 bits) storing **obf_enc_key** and **obf_enc_iv** used for obfuscation. Telegram calls this simply *bytes*.
- **obf_enc_key** The key used for encryption to obfuscate data.
- **obf_enc_iv** The IV used for encryption to obfuscate data.

- **obf_dec_key_bytes** 64 bytes derived from **obf_enc_key_bytes** storing the server's encryption key and IV. Labeled in Telegram as *temp*.

3.4.1.1 Temporary encryption key

The process starts by generating 64 random bytes `obf_enc_key_bytes`. The first 8 bytes are unused. Bytes 8 – 39 are used as an encryption key and bytes 40 – 55 as an IV. Bytes 56 – 63 are composed of the last 8 bytes of `obf_enc_key_bytes` encryption of itself. It is unclear what are those bytes used for. The final `obf_enc_key_bytes` to be sent is visualized in Figure 3.4.

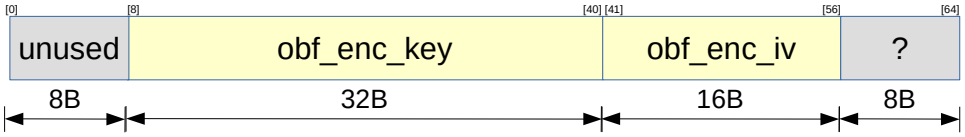


Figure 3.4: The `obf_enc_key_bytes` containing random bytes and its usage for the obfuscation encryption.

The length of the packet, yet again encrypted, and the real data to be transmitted as expected from the official documentation are then AES-CTR encrypted using the `obf_enc_key` and `obf_enc_iv`, and sent. All the other data in this TCP stream are encrypted using the same obfuscation key. When another connection is established, a new `obf_enc_key_bytes` is generated and the process repeats itself.

3.4.1.2 Temporary decryption key

Besides the `obf_enc_key_bytes` setup, the very same function deals with setting up the `obf_dec_key_bytes`. This temporary key is used for decrypting the incoming traffic.

The `obf_dec_key_bytes` is derived from the `obf_enc_key_bytes`. 48 bytes are reversed from `obf_enc_key_bytes`, starting at position 8. This may be seen in Listing 3. The first 32 bytes are then used as a key and the next 16 bytes as an IV for the incoming traffic decryption.

```
for (int a = 0; a < 48; a++) {
    obf_dec_key_bytes[a] = obf_enc_key_bytes[55 - a];
}
```

Listing 3: Temporary decryption key deduction from the `obf_enc_key_bytes` array. First 32 bytes are used as a decryption key, next 16 bytes for an IV. Telegram for Android source code, file `Connections.cpp`, line 331.

We believe Telegram server receives the `obf_enc_key_bytes`, tampers with them in a way described in this section and finally encrypts the response with `obf_dec_key_bytes`. As mentioned earlier, this is not officially documented whatsoever. A more conventional approach would be much appreciated, such as the usage of SSL/TLS – a cryptographic protocol scrutinized thoroughly by researchers all over the world.

To sum up, the whole function is depicted in Listing 4 in its simplified version. It also uses our own terminology to comply with this section.

```
void sendData(payload)
{
    obf_enc_key_bytes = byte[64];

    if (!firstPacketSent) {
        obf_dec_key_bytes = byte[64];
        fillWithRandom(obf_enc_key_bytes);

        for (int a = 0; a < 48; a++) {
            obf_dec_key_bytes[a] = obf_enc_key_bytes[55 - a];
        }

        setAESEncryptKey(obf_enc_key_bytes + 8);
        setEncryptIv(obf_enc_key_bytes + 40);

        setAESDecryptKey(obf_dec_key_bytes);
        setDecryptIv(obf_dec_key_bytes + 32);

        send(obf_enc_key_bytes);
        firstPacketSent = true;
    }

    send(AESCTREncrypt(packetLength));
    send(AESCTREncrypt(payload))
}
```

Listing 4: The function starts by generating random bytes. The decryption key is then derived, and both encrypt and decrypt keys are set. Finally, the length of the payload (obfuscated), the `obf_enc_key_bytes` and the actual IGE encrypted payload are sent. The function's argument – the payload – is in the form expected by Figure 3.3. Telegram for Android source code, file `Connection.cpp`, line 289, redacted.

3.4.2 Deobfuscation program

To verify these findings and to continue the analysis we created a software to deobfuscate the traffic. Even though the first choice of language was Python to keep all the scripts in one package, we finally opted for C++. This has two major advantages: one, we can be directly inspired by the Telegram code, and second, we can access OpenSSL functions directly the same way Telegram does[17].

The program is capable of fetching the obfuscation key and transforming the data to its deobfuscated form and to further analyze it. It takes two or three arguments as an input:

- **incoming stream** sniffed incoming data
- **outgoing stream** sniffed outgoing data
- **key** file containing the user's `auth_key` (optional)

The first and second argument are binary files of the sniffed traffic. Since the `obf_enc_key_bytes` value is sent by the client, it is stored in the outgoing stream, and it is therefore not possible to deobfuscate any traffic without these data. We used Wireshark repeatedly to follow the TCP stream and then saved both the incoming and outgoing data into a binary file. We then ran the program with those files which proved to be a viable technique.

The third argument is optional. It is the user's secret `auth_key`. This is obviously not available to an attacker, but it is helpful for study purposes. When extracted, the user may see the real traffic Telegram generates and what actions are taken.

Readme file is present describing how to build and run the program.

3.5 Replay attack

During the analysis we examined some parts of the code in an attempt to discover potential vulnerabilities to exploit. To provide a little bit of a context, let us now briefly describe how Telegram processes all incoming data.

3.5.1 Incoming data processing

The Figure 3.5 shows how Telegram handles all incoming data. First, the `Connection::onReceivedData()` function is called, and incoming data are deobfuscated in the way described in Section 3.4.

Next, the `ConnectionsManager::onConnectionDataReceived()` function is called. If the `auth_key_id` is not set to 0 (which is only the case during

[17]Telegram actually uses BoringSSL – a fork of OpenSSL by Google but for our case the distinction was not relevant.

the key exchange process; see Section 2.3.1 for more details), the function `Datacenter::decryptServerResponse()` is invoked. This function checks the `auth_key_id` and if valid, decrypts the actual payload using the master secret `auth_key` and the derived `msg_key`. Afterwards, the message ID and other fields are verified.

If the decryption succeeds, the `onConnectionDataReceived()` function proceeds to further process the decrypted payload, and several other checks are performed. Finally, if all checks succeed, the whole process is finalized by the `ConnectionsManager::processServerResponse` function.

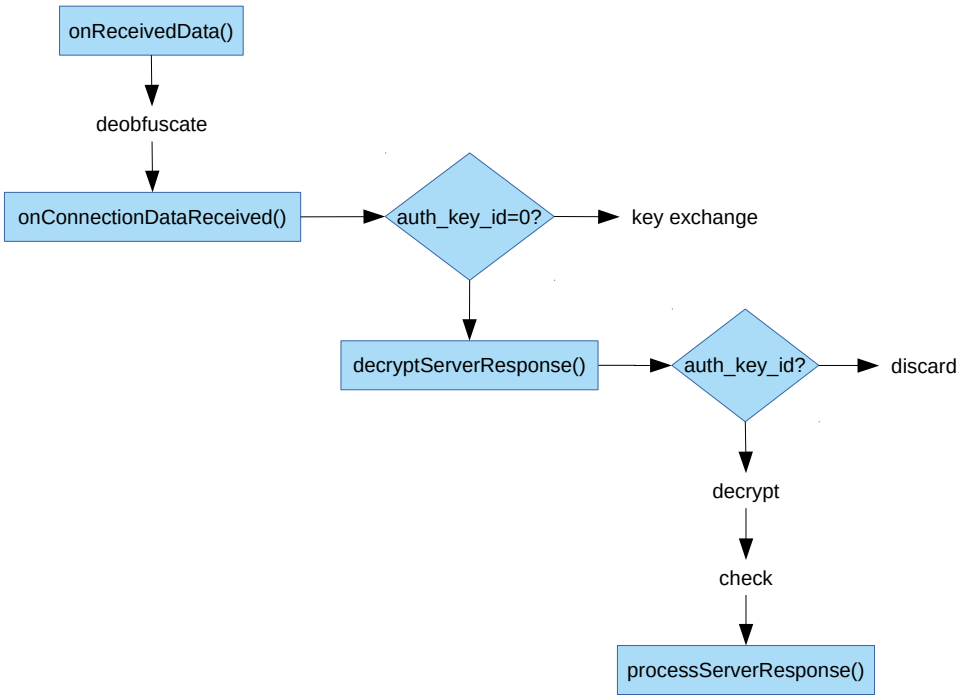


Figure 3.5: The incoming data are first deobfuscated. If the non-zero `auth_key_id` is valid, the message gets decrypted and checked for validity. The process is completed by the `processServerResponse()` function.

3.5.2 Vulnerability

A replay attack is an attack where an attacker sniffs data sent by the application and then maliciously resends them at a different time. By doing so, the attacker can repeat any message without the user noticing; curiously, the attacker does not actually know the message. A protection might be realized by implementing a message counter to keep track of the order the messages appear in.

We analyzed how Telegram deals with this issue, see Listing 5. After the decryption it checks if the message was already processed. The function `ConnectionSession` holds an internal array of the already processed messages. If the message is accepted and processed, the message ID is then added into the array.

```
if (connection->isMessageIdProcessed(messageId)) {
    doNotProcess = true;
}

if (!doNotProcess) {
    // process
    addProcessedMessageId(messageId);
}
```

Listing 5: Each incoming message is checked whether it was already processed. If not, the message is further processed and finally marked as such.
Telegram for Android source code, file `ConnectionsManager.cpp`, line 728.

The behavior we believe might be exploitable concerns the `addProcessed MessageId()` function. The function checks the size of the array and if it exceeds 300, it erases the first 100 messages as seen in Listing 6.

```
void addProcessedMessageId(messageId)
{
    if (processedMessageIds.size() > 300) {
        processedMessageIds.erase(processedMessageIds.begin(),
            processedMessageIds.begin() + 100);
    }
    processedMessageIds.push_back(messageId);
}
```

Listing 6: After message is successfully processed its ID is added to an internal array of processed messages.
Telegram for Android source code, file `ConnectionSession.cpp`, line 55.

The attack scenario is drafted as follows:

1. Sniff a message
2. Wait for 300 other messages
3. Replace the next message with the first one

4. Telegram processes the first message again

If Telegram does not provide any additional checks and actually deletes the first 100 IDs, the message would be accepted twice.

With the attack scenario drafted we attempted to perform the attack.

3.5.3 Exploit attempt

To perform the attack we first decided to inject a new TCP packet containing the payload copied from the very first packet. That proved to be rather difficult since it is not possible to inject a new TCP packet without interfering with the TCP connection. We finally opted for replacing the payload of the next message (i.e. the 301st), meaning we would drop its payload and replace it with the first one. That should result in the victim receiving the first message again instead of the new one.

3.5.3.1 Required tools

As we found out, neither Scapy nor Wireshark are the right tools for this task. Both are useful for passive sniffing, and while Scapy is capable of sending additional packets, it does not allow for a modification of the packets in real time as it does not reside in the middle of the traffic as described in the Setup, Section 3.1.

Furthermore, since Telegram uses its own Application layer protocol, we were unable to use popular tools such as Fiddler, Burp Suite or OWASP ZAP which are built for HTTP [56, 57].

Ettercap, already mentioned earlier to perform the MitM, provides a simple filtering interface allowing to modify the data routed through. We tried this approach but Ettercap requires the filters to be written in its own language which is very limiting and unfortunately insufficient for our needs.

The scenario was finally tested with the Trudy software which can modify any TCP traffic. As its documentation states, Trudy is *"a transparent proxy that can modify and drop traffic for arbitrary TCP connections"* [58]. All traffic is routed through Trudy which then applies so called *modules*. Trudy modules are bulks of preprogrammed code to perform modifications desired by the user. Trudy provides an example stub of such a module, and users are encouraged to implement it. Because Trudy is written in Go, all modules are to be written in Go as well. To route the traffic and setup the environment properly, a prepared virtual machine is available [59].

After setting up the MitM as described in 3.1, a virtual machine is run on the computer using Vagrant[18]. The VM installs all the required software and Trudy itself as well. Then it routes all the traffic in and out of Trudy using

[18]Vagrant is a simple wrapper around a virtualization provider for building completely independent virtual environments.

`iptables`. Trudy receives all the traffic, modifies it based on the used module and sends it back to internet as seen in Figure 3.6.

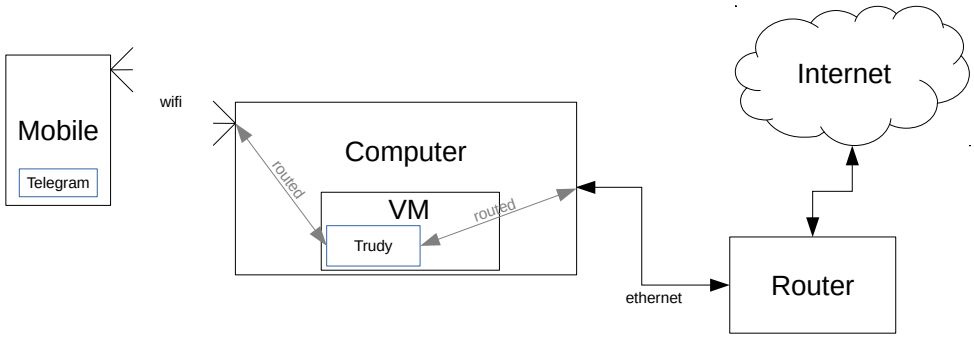


Figure 3.6: The computer runs Trudy inside a virtual machine. All traffic is routed bidirectionally through Trudy.

The implementation of a module for Trudy is quite straightforward. Trudy calls specific methods in fixed order, each designed for one particular action. The methods of a single module are depicted in 3.7 and a description of the methods follows.

- **Deserialize()** converts the raw payload into a known structure of data (e.g. HTTP)
- **Drop()** if *true* is returned, the whole packet is discarded
- **DoMangle()** if *true* is returned, `Mangle()` is called
- **Mangle()** alters the payload
- **DoIntercept()** if *true* is returned, the data are sent to the Trudy interceptor[19]
- **DoPrint()** if *true* is returned, `PrettyPrint()` is called
- **PrettyPrint()** prints data in a human friendly format
- **Serialize()** converts the data back to raw payload if it were previously deserialized
- **BeforeWriteTo()** actions taken before the data are written to one side or the other
- **AfterWriteTo()** actions taken after the data are written to one side or the other

[19]Trudy interceptor is a simple local website the user may use to interfere with the data. It was not investigated further.

3.5.3.2 Preliminaries

To test our settings we didn't wait for the next 300 messages; instead we set up breakpoints on the very lines where Telegram checks if the ID was already processed or not, the line 729 and 859 in the `ConnectionsManager` file in particular. If the packet was actually send again, these breakpoints would be fired, and the message further rejected by Telegram. This would confirm our surmise and later be adjusted to comply fully with the scenario.

To further simplify the process we limited the test messages to the length of 201 bytes to easily distinguish a message from the other traffic. The length of the message remained the same throughout the testing, we only changed some of the characters each time, to see which messages arrived. The testing message we used reads:

> *"To simplify the process we are using a message longer than 200 bytes to easily identify it in the stream of data"*

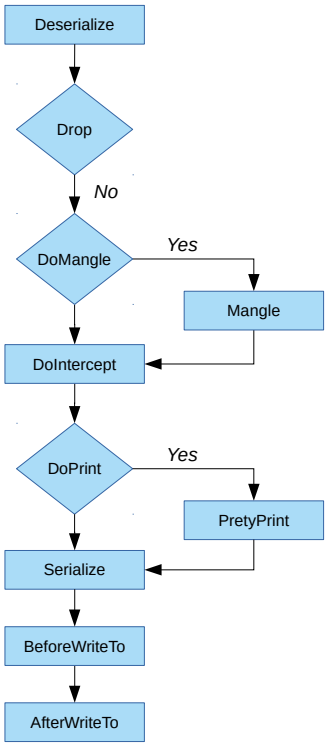


Figure 3.7: Trudy module's functions are called in a predefined fixed order.

When sending such message using Telegram, its length is always equal to 201 bytes under regular conditions.

3.5.3.3 Execution

We implemented the Trudy module as shown in Listing 7. The `DoMangle()` function checks if the source IP address is equal to one of the Telegram datacenters (`149.154.167.91:80`). We confirmed our testing application is communicating with this server using Wireshark, and also the configuration details extracted from the `tgnet.dat` file (see Section 3.3.1) contained a datacenter with such IP address. Furthermore, the size of the packet is checked to select the testing messages only.

If those conditions are met, the `Mangle()` function checks whether some bytes were already saved. If not, it signifies this is the first message and it copies the TCP payload into the `oldBytes` array and sets the `saved` flag to

true. It does not modify the payload in any way. If `saved` is already set to *true*, it copies the saved bytes into the currently intercepted TCP packet, replacing its content. Keep in mind, we are attempting the simplified scenario and therefore not waiting for 300 other messages, we are trying to resend the message as soon as another one comes in, which should fire the breakpoints.

```
var saved bool
var oldBytes []byte

func (input Data) DoMangle() bool {
        if input.ServerAddr.String() == "149.154.167.91:80"
                && len(input.Bytes) == 201 {
                return true
        }
        return false
}

func (input *Data) Mangle() {
        if saved {
                copy(input.Bytes, oldBytes)
        } else {
                oldBytes = make([]byte, len(input.Bytes))
                copy(oldBytes, input.Bytes)
                saved = true
        }
}
```

Listing 7: The programmed Trudy module code written in Go used to perform a Replay attack on Telegram. The `DoMangle()` function limits the packet modifications only to our messages. `Mangle()` actually performs the attack.

3.5.3.4 Results and Future Work

Unfortunately, that was not the case. We confirmed Telegram successfully receives the repeated traffic but none of the desired breakpoints were fired. Further analysis showed that Telegram incorrectly deobfuscated the traffic. This is due to the fact that during the communication the `obf_enc_key_bytes` changes periodically. The deobfuscated data are therefore completely different and are rejected in various places as nonsense.

We were unable to continue the attack due to time limitations and the finite scope of this work. However, we still believe this attack is feasible. In

order to continue the obfuscation keys would have to be saved as well. The modified scenario goes as follow:

1. Sniff a message, deobfuscate it and save it
2. Wait for 300 other messages
3. Save the current obfuscation key
4. Replace the next message with the first one obfuscated by the current key
5. Telegram processes the first message again

This scenario requires the module to have the obfuscation capabilities introduced in the C program in Section 3.4.2. Since Go has a native support of C, the codebase could be incorporated into the Trudy module.

3.5.3.5 Responsible Disclosure

The findings were reported to the Telegram security team on December 7th, 2016 with a kind request for comments. Two points were discussed in particular, the obfuscation method and the Replay attack scenario. The first response from Telegram was received on 12th December 2016.

The obfuscation method was commented only briefly as *"unrelated to data security and is used to counter some of the less sophisticated attempts at banning our service in certain countries"*. In October 2015, Pavel Durov (the founder of Telegram) stated on his Twitter account that Telegram was blocked temporarily in Iran as a result of Telegram refusing to collaborate with the Iranian government [60]. This most likely concerns other countries applying some form of internet censorship as well but illustrates well that Telegram indeed faces censorship issues.

MTProto has a fixed structure where the `auth_key_id` value is always present at the beginning of the packet and therefore easily recognizable, which may be considered as a design flaw of the protocol itself. Another solution to this might be wrapping the protocol into SSL/TLS. The traffic would be indistinguishable from other protocols based on SSL/TLS, such as the very common HTTPS, making it even harder to identify. Telegram developers dismissed this proposal claiming SSL/TLS too performance heavy.

Under these circumstances it is understandable why the obfuscation method is not officially documented in any form.

Secondly, Telegram responded to our attack scenario from Section 3.5 and actually accepted some of our remarks. According to the Telegram team the Android application (as opposed, allegedly, to the other clients, such as Telegram for iOS, Telegram Desktop etc.) does not perform one of the the security

checks as is required by the protocol, defined in the Security Guidelines for Client Developers in particular [61].

Among other checks, the Security Guidelines require [61] the message ID to be checked against the stored ones and Telegram performs that. However, it requires one more additional check – if the incoming message has an ID lower than all or equal to any of the stored IDs, such message is to be discarded. This action does indeed diminish the risk but Telegram for Android did not carry out this check.

The Telegram team further commented that this vulnerability does not allow the attacker to cause any severe damage because of the additional protection on the side of the Telegram API. Message actions (sending, editing, deleting, and changing read status), group membership, secret chats, and other important areas are not affected[20]. Nevertheless, the Telegram team confirmed the attack would work for nonessential service updates like online or typing statuses. For example, the scenario would allow the attacker to alter the statuses of victim's friends (as seen in the Android application, not in the Telegram network) or spoof the victim to see typing statuses from contacts not performing such action in reality.

We briefly reviewed these claims and concluded that the Java part of the Telegram application does indeed deal with additional identifiers, such as `qts`, `pts` and others. These values do seem to provide additional protection. It is important to mention that we did not perform any deeper research.

Telegram promised this will be fixed in the next Android update – most likely in the 3.16 version which should be released in early January 2017. Telegram also offered a financial reward for our findings.

[20] According to Telegram, this is because of the checks done in the `MessagesController.java` class on lines 5731, 5561, 5765, 5817 and others.

Conclusion

In the scope of this thesis our objective was to describe a selection of currently used Instant Messenger solutions and discuss their security aspects. Furthermore, the thesis aims to investigate the Telegram Instant Messenger in more detail, mainly its homebrew protocol MTProto and its code.

We have analyzed many security-related incidents and presented their impact on the end user. We have indicated how the messengers respond to such incidents, and whether they try to improve on their security. A number of security-related problems still remain. Many messengers are closed-source, thus not providing an opportunity for an independent code review, and insufficient or completely missing documentation raises severe questions as well.

Moreover, we have studied the Telegram IM and its MTProto protocol. We have thoroughly documented the protocol's internal working, its initialization and the encryption process. Since Telegram has two types of chat environment (regular and secret chats), we have covered both and stressed out how they relate to each other. We have noted the cryptographic primitives it uses as well.

We set up a testing laboratory using various tools and analyzed the network traffic Telegram produces. We have scrutinized the code base of the official application for Android and concluded that the state of the application is at serious odds with the documentation. This concerns mainly the undocumented obfuscation method Telegram uses. The MTProto traffic is encrypted one more time with the key and IV prepended to the data. This has no effect on the data security and is easily debunked by the deobfuscation program we have implemented. When the Telegram team was confronted with these claims, they noted the method is used to circumvent some of the less sophisticated methods of censorship in certain countries.

Finally, we have localized an exploitable vulnerability and drafted an attack scenario. We concluded that the Android application does not check the message identification numbers properly and that a Replay attack might be feasible. Although our primary scenario of the attack turned out not to be ap-

plicable, we have drafted an altered scenario which we believe would work. We have also reported our findings to the Telegram security team which accepted our remarks and agreed, to a certain degree, that this might be exploitable. Telegram promised to fix this issue in the next software release.

Our work mainly focuses on the protocol and the Android client, but there are still many areas where further research might be required. Additional research might focus on the other Telegram clients, such as the desktop or iOS version, studying the protocol as a whole, or administering other forms of attacks.

Bibliography

[1] Ahmed, M. *Telegram hits 100m users and commits to remaining ad-free* [online]. Feb. 2016, [accessed 2016-12-16]. Available from: https://www.ft.com/content/de54c280-d97d-11e5-a72f-1e7744c66818

[2] Cox, J. *Encryption App Telegram Probably Isn't as Secure for Terrorists as ISIS Thinks* [online]. Nov. 2015, [accessed 2016-12-16]. Available from: https://motherboard.vice.com/read/encryption-app-telegram-probably-isnt-as-secure-for-terrorists-as-isis-thinks

[3] Leyden, J. *Homebrew crypto in Telegram hangout app full of holes, say security pros* [online]. Nov. 2015, [accessed 2016-12-16]. Available from: http://www.theregister.co.uk/2015/11/23/homebrew_crypto_in_telegram_app/

[4] Covert, A. *Facebook buys WhatsApp for $19 billion* [online]. Feb. 2014, [accessed 2016-01-26]. Available from: http://money.cnn.com/2014/02/19/technology/social/facebook-whatsapp/

[5] BBC. *WhatsApp reaches a billion monthly users* [online]. Feb. 2016, [accessed 2016-02-02]. Available from: http://www.bbc.com/news/technology-35459812

[6] Chen, B. *Worried About the Privacy of Your Messages? Download Signal* [online]. Dec. 2016, [accessed 2016-01-02]. Available from: http://www.nytimes.com/2016/12/07/technology/personaltech/worried-about-the-privacy-of-your-messages-download-signal.html

[7] Custer, C. *WeChat blasts past 700 million monthly active users, tops China's most popular apps* [online]. Aug. 2015, [accessed 2016-06-23]. Available from: https://www.techinasia.co/wechat-blasts-700-million-monthly-active-users-tops-chinas-popular-apps

[8] Electronic Frontier Foundation. *Secure Messaging Scorecard* [online]. [accessed 2016-02-02]. Available from: https://www.eff.org/secure-messaging-scorecard

[9] WhatsApp. *WhatsApp homepage* [online]. Jan. 2016, [accessed 2016-01-26]. Available from: https://www.whatsapp.com/?l=en

[10] Kurtz, A. *Shooting the Messenger* [online]. Sept. 2011, [accessed 2016-01-31]. Available from: http://www.andreas-kurtz.de/2011/09/shooting-messenger.html

[11] Gevers, R. *Hijack Whatsapp with your iPhone* [online]. Sept. 2011, [accessed 2016-01-31]. Available from: http://rickey-g.blogspot.com/2011/05/hijack-someone-elses-whatsapp-with-your.html

[12] mgp25. *WhatsApp Protocol - FunXMPP* [online]. Dec. 2014, [accessed 2016-01-31]. Available from: https://github.com/mgp25/Chat-API/wiki/FunXMPP-Protocol

[13] Heckel, P. C. *How To: Sniff the WhatsApp password from your Android phone or iPhone* [online]. July 2013, [accessed 2016-01-31]. Available from: https://blog.heckel.xyz/2013/07/05/how-to-sniff-the-whatsapp-password-from-your-android-phone-or-iphone/

[14] Damania, D. *Use Whatsapp? You Phone number is your Username and IMEI is the password – Hackable* [online]. Sept. 2012, [accessed 2016-01-31]. Available from: http://thednetworks.com/2012/09/09/whatsapp-imei-password-md5-inverted-hack/

[15] Yourdailymac. *WhatsApp leaks usernames, telephone numbers and messages* [online]. May 2011, [accessed 2016-02-02]. Available from: http://www.yourdailymac.net/2011/05/whatsapp-leaks-usernames-telephone-numbers-and-messages/

[16] Summerson, C. *WhatsAppSniffer Shames WhatsApp's Plaintext, Unprotected Chat Transfer Protocol, Shows Off Just How Much Can Be Sniffed* [online]. May 2012, [accessed 2016-02-02]. Available from: http://www.androidpolice.com/2012/05/02/whatsappsniffer-shames-whatsapps-plaintext-unprotected-chat-transfer-protocol-shows-off-just-how-much-can-be-sniffed/

[17] djwm. *Sniffer tool displays other people's WhatsApp messages* [online]. May 2012, [accessed 2016-02-02]. Available from: http://www.h-online.com/security/news/item/Sniffer-tool-displays-other-people-s-WhatsApp-messages-1574382.html

[18] fab. *WhatsApp no longer sends plain text* [online]. Aug. 2012, [accessed 2016-02-02]. Available from: http://www.h-online.com/security/news/item/WhatsApp-no-longer-sends-plain-text-1674723.html

[19] Alkemade, T. *Piercing Through WhatsApp's Encryption* [online]. Oct. 2013, [accessed 2016-02-02]. Available from: https://blog.thijsalkema.de/blog/2013/10/08/piercing-through-whatsapp-s-encryption/

[20] Brewster, T. *WhatsApp Users 'Should Not Trust Broken Encryption'* [online]. Oct. 2013, [accessed 2016-02-02]. Available from: http://www.techweekeurope.co.uk/workspace/whatsapp-encryption-security-128964

[21] Marlinspike, M. *Open Whisper Systems partners with WhatsApp to provide end-to-end encryption* [online]. Nov. 2014, [accessed 2016-01-26]. Available from: https://whispersystems.org/blog/whatsapp/

[22] WhatsApp. *WhatsApp FAQ* [online]. [accessed 2016-02-02]. Available from: https://www.whatsapp.com/faq/en/general/21864047

[23] Scherschel, F. *Keeping Tabs on WhatsApp's Encryption* [online]. Apr. 2015, [accessed 2016-02-02]. Available from: http://www.heise.de/ct/artikel/Keeping-Tabs-on-WhatsApp-s-Encryption-2630361.html

[24] WhatsApp. *WhatsApp Encryption Overview* [online]. Apr. 2016, [accessed 2016-12-16]. Available from: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf

[25] Frosch, T.; Mainka, C.; Bader, C.; et al. *How Secure is TextSecure?* Cryptology ePrint Archive, Report 2014/904, Oct. 2014. Available from: https://eprint.iacr.org2/014/904

[26] Appelbaum, J.; Gibson, A.; Grothoff, C.; et al. *Prying Eyes: Inside the NSA's War on Internet Security* [online]. Dec. 2014, [accessed 2017-01-02]. Available from: http://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html

[27] Franceschi-Bicchierai, L. *Snowden's Favorite Chat App Is Coming to Your Computer* [online]. Dec. 2015, [accessed 2017-01-02]. Available from: https://motherboard.vice.com/rea/signal-snowdens-favorite-chat-app-is-coming-to-your-computer

[28] McCormick, R. *Edward Snowden's favorite encrypted chat app is now on Android* [online]. Nov. 2015, [accessed 2016-02-10]. Available from: http://www.theverge.com/2015/11/3/9662724/signal-encrypted-chat-app-android-edward-snowden

[29] Threema GmbH. *How does Threema audit its code?* [online]. Aug. 2015, [accessed 2017-01-02]. Available from: https://threema.ch/en/faq/code_audit/

[30] Branigan, T. *China blocks Twitter, Flickr and Hotmail ahead of Tiananmen anniversary* [online]. June 2009, [accessed 2017-01-02]. Available from: https://www.theguardian.com/technology/2009/jun/02/twitter-china

[31] McKerrow, K. *Facebook's Zuckerberg Still Wants to Conquer China* [online]. June 2016, [accessed 2016-06-23]. Available from: https://www.thestreet.com/story/13615119/1/facebook-s-zuckerberg-still-wants-to-conquer-china.html

[32] Branigan, T. *China intensifies crackdown on social media with curbs on instant messaging* [online]. Aug. 2014, [accessed 2016-06-23]. Available from: https://www.theguardian.com/world/2014/aug/07/china-intensifies-social-media-crackdown-curbs-instanrt-messaging

[33] Levin, N. *China Tightens Restrictions on Messaging Apps* [online]. Aug. 2014, [accessed 2016-06-23]. Available from: http://www.wsj.com/articles/china-issues-new-restrictions-on-messaging-apps-1407405666

[34] Davison, N. *WeChat: the Chinese social media app that has dissidents worried* [online]. Dec. 2012, [accessed 2016-06-23]. Available from: https://www.theguardian.com/world/2012/dec/07/wechat-chinese-social-media-app

[35] Xiao, C. *More Details on the XcodeGhost Malware and Affected iOS Apps* [online]. Sept. 2015, [accessed 2016-06-23]. Available from: http://researchcenter.paloaltonetworks.com/2015/09/more-details-on-the-xcodeghost-malware-and-affected-ios-apps/

[36] Telegram. *Telegram FAQ* [online]. [accessed 2016-01-26]. Available from: https://telegram.org/faq

[37] Petrielli, P. *Telegram Messenger: Review* [online]. July 2015, [accessed 2016-02-02]. Available from: http://techglobule.com/2015/07/telegram-messenger/

[38] Ruvolo, J. *Brazilian Judge Shuts Down WhatsApp And Brazil's Congress Wants To Shut Down The Social Web* [online]. Dec. 2015, [accessed 2016-01-26]. Available from: http://www.techcrunch.com/2015/12/16/brazils-congress-has-shut-down-whatsapp-tonight-and-the-rest-of-the-social-web-could-be-next

[39] Savov, V. *Brazil's WhatsApp ban is driving millions of users to Telegram* [online]. Dec. 2015, [accessed 2016-02-02]. Available from: http://www.theverge.com/2015/12/17/10386776/brazil-whatsapp-ban-telegram-millions-users

[40] Jacobs, F. *On SMS logins: an example from Telegram in Iran* [online]. Apr. 2016, [accessed 2016-06-23]. Available from: https://www.fredericjacobs.com/blog/2016/01/14/sms-login/

[41] Jacobs, F. *On SMS Logins II : an example from Telegram in Russia* [online]. Apr. 2016, [accessed 2016-06-23]. Available from: https://www.fredericjacobs.com/blog/2016/04/30/more-on-sms-logins/

[42] Marlinspike, M. *A Crypto Challenge For The Telegram Developers* [online]. Dec. 2013, [accessed 2016-01-26]. Available from: http://thoughtcrime.org/blog/telegram-crypto-challenge/

[43] Crypto Fails. *Telegram's Cryptanalysis Contest* [online]. Dec. 2013, [accessed 2016-01-26]. Available from: http://www.cryptofails.com/post/70546720222/telegrams-cryptanalysis-contest

[44] Jakobsen, J. B. *A practical cryptanalysis of the Telegram messaging protocol* [online]. Sept. 2015. Available from: http://cs.au.dk/~jakjak/master-thesis.pdf

[45] Telegram. *FAQ for the Technically Inclined* [online]. [accessed 2016-10-07]. Available from: https://core.telegram.org/techfaq

[46] Paar, C.; Pelzl, J.; Preneel, B. *Understanding cryptography: a textbook for students and practitioners*. Springer, 2010, ISBN 978-3-642-04100-6.

[47] Laurie, B. *OpenSSL's Implementation of Infinite Garble Extension* [online]. Aug. 2006, [accessed 2016-07-09]. Available from: http://www.links.org/files/openssl-ige.pdf

[48] Schneier, B. *SHA-1 Broken* [online]. Feb. 2005, [accessed 2016-02-09]. Available from: https://www.schneier.com/blog/archives/2005/02/sha1_broken.html

[49] Klyubin, A. *Some SecureRandom Thoughts* [online]. Aug. 2013, [accessed 2016-07-09]. Available from: https://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html

[50] adc. *A 2^{64} Attack On Telegram, And Why A Super Villain Doesn't Need It To Read Your Telegram Chats.* [online]. Jan. 2015, [accessed 2016-02-09]. Available from: http://www.alexrad.me/discourse/a-264-attack-on-telegram-and-why-a-super-villain-doesnt-need-it-to-read-your-telegram-chats.html

[51] Google Inc. *Android Studio* [online]. [accessed 2016-11-09]. Available from: https://developer.android.com/studio/index.html

[52] Haight, J. *Spoofing Wifi with a Pineapple* [online]. Feb. 2015, [accessed 2016-11-28]. Available from: https://www.psattack.com/articles/20150212/spoofing-wifi-with-a-pineapple/

[53] Ettercap. *Ettercap homepage* [online]. [accessed 2016-11-28]. Available from: https://ettercap.github.io/ettercap/

[54] DrKLO. *Telegram messenger for Android* [online]. Oct. 2013, [accessed 2016-11-01]. Available from: https://github.com/DrKLO/Telegram

[55] Google Inc. *Improve Your Code with Lint* [online]. [accessed 2016-11-11]. Available from: https://developer.android.com/studio/write/lint.html

[56] Telerik. *Fiddler homepage* [online]. [accessed 2016-12-03]. Available from: http://www.telerik.com/fiddler

[57] Portswigger. *Burpsuite homepage* [online]. [accessed 2016-12-03]. Available from: https://portswigger.net/burp/

[58] Ludwig, K. *Trudy* [online]. [accessed 2016-12-05]. Available from: https://github.com/praetorian-inc/trudy

[59] Ludwig, K. *Trudy VM* [online]. [accessed 2016-12-05]. Available from: https://github.com/praetorian-inc/mitm-vm

[60] Durov, P. *Pavel Durov's Twitter account* [online]. Oct. 2015, [accessed 2016-12-14]. Available from: https://twitter.com/durov/status/656551981226528768

[61] Telegram. *Security Guidelines for Client Developers* [online]. [accessed 2016-12-14]. Available from: https://core.telegram.org/mtproto/security_guidelines#checking-msg-id

Appendix A

Contents of CD

- `readme.txt` ..... the file with CD contents description
- `data` ..... the working data files directory
  - `Telegram` ..... Telegram 3.13.1 source code
- `src` ..... the directory of source codes
  - `Telegram Deobfuscator` .. Deobfuscator source code; see Section 3.4.2
  - `Telegram Extractor` ..... Extractor source code; see Section 3.3.1
  - `Trudy module` ..... written Trudy module; see Section 3.5.3.3
  - `thesis` ..... LaTeX source codes of the thesis
  - `diagrams` ..... illustrations source code
- `text` ..... this thesis text directory
  - `appendices` ..... thesis appendices as mentioned in the text
  - `thesis.pdf` ..... this thesis in PDF format