Abstract

Telegram is a popular instant messaging service, a self-described fast and secure solution. It introduces its own home-made cryptographic protocol MTProto instead of using already known solutions, which was criticised by a significant part of the cryptographic community [2, 8]. In this article we will briefly introduce the protocol to provide context to the reader and then present two major findings we discovered as part of our security analysis performed in late 2016. First, the undocumented obfuscation method Telegram uses, and second, a replay attack vulnerability we discovered. The analysis was mainly focused on the MTProto protocol and the Telegram’s official client for Android.

1 Introduction

Nowadays, the human interaction and communication take place more and more often in the digital realm and instant messaging became the digital version of a spoken dialogue. The exposure of mass surveillance leaked by the ex-NSA employee Edward Snowden fuelled the demand for a secure and privacy-aware communication solutions.

One of such proposed solutions is Telegram. Telegram is an instant messaging service enabling users to send messages, photos, videos, stickers, files and lately even voice calls. Telegram describes itself as a fast and secure solution for instant messaging and claims to be safer than WhatsApp. It 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 provides two modes of messaging: the regular chat and the 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 [15]. For both services the MTProto protocol is used; secret chats only apply yet another layer of protection. Since the choice of mode is irrelevant to this article, we will focus on the regular chats only.

2 Cryptography behind Telegram

Telegram authors decided to craft a brand new encryption protocol, 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 a research performed at Aarhus University in spring 2015 [6].

2.1 Initialization

During the first launch of the Telegram application, the 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 by doing so verifies the phone number. When this process is done, the registration process begins.

The gist of the registration process is to create a secret key called auth_key which is then used for all client-server communication and regular chats. Next, a fingerprint of this key, labeled as auth_key_id, is created. It is crafted from the last 64 bits of SHA1(auth_key).

We are omitting the details here, however we reference the full analysis report [13] should the reader be inclined to find out more.

2.2 Payload

Let us describe the content of a single message further referred to as a payload. The payload prior to encryption is depicted in Figure 1 and it contains:

- salt Periodically changed number used for various protection purposes.
- session_id Unique number to identify the user and their device.
Figure 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.

- **msg_id** Unique ID of the message within a session.
- **seq_no** Message sequence counter.
- **length** Length of the actual message.
- **message** Content of the actual message.

### 2.3 Encryption

The whole encryption process is visualized in Figure 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 $auth_key$ obtained during the initialization process 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 $auth_key_id$. The data array is then ready to be transported to the server.

#### 2.3.1 AES and IGE block cipher mode

For the real encryption process the Advanced Encryption Standard (AES) is used alongside with the Infinite Garble Extension (IGE) block cipher mode. IGE is a lesser-known block cipher mode shown on Figure 3. It is defined by the following formula [7]:

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

where $f_K$ stands for the encryption 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 [7].

The reasons behind the choice of IGE are unclear.

### 2.4 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.

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 4 and 5.
2.5 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. All data are considered as an input into the MTProto protocol for regular chats.

Secret chats have a number of additional fields in its payload and there are some other minor differences, all quite insignificant for the essence of this article, hence are omitted here.

3 Analysis

The official Android application is publicly available on Github [3] and our research involved the latest (as of October 2016) Telegram version 3.13.1, the 64e8ec3 commit in particular.

The application’s main programming language is Java but it uses a Java Native Interface (JNI) to incorporate C libraries such as boringssl, ffmpeg, sqlite and also Telegram’s own C library called tgnet. This library is responsible for many connection-related actions including communication with the server and sending of messages. The secret messages are still dealt with within the Java part of the application.

We commenced our analysis by installing Telegram for Android and analyzed it with various tools in order to draft a comparison between the official documentation and other sources.

4 Part I: Undocumented obfuscation

After setting up our research environment we wanted to sniff the data sent from the application. During this data collection we found that the received data did not correspond to the documentation at all. We expected the data to be in a form of auth_key_id, msg_key and encrypted data as per Figure 6.

Using various scripts of our design we extracted 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 [6].

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. Rather than sending the data in the expected manner, it is encrypted one more time with a random key sent 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.

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 the obfuscating encryption.

• **obf_enc_iv**: The IV used for the obfuscating encryption.

• **obf_dec_key_bytes**: 64 bytes derived from **obf_enc_key_bytes**, containing the server’s encryption key and IV. Labeled in Telegram as **temp**.

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 7.

![Figure 7](image)

The length of the packet, yet again encrypted, and the real data to be transmitted according to 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.

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 1. The first 32 bytes are then used as a key and the next 16 bytes as an IV for the incoming traffic decryption.

We believe the 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 in any way.

To sum up, the whole function is depicted in Listing 2 in its simplified version. It also uses our own terminology to comply with this section.

```cpp
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];
        }
        setAEEncryptKey(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 2: 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 6. Telegram for Android source code, file `Connections.cpp`, line 289, redacted.

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, we finally opted for C++. This had two major advantages: first, we can be directly inspired by the Telegram code, and second, we can access OpenSSL functions directly the same way Telegram does.\footnote{1}

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.

We also wrote number of scripts to extract miscellaneous data from the client. See Section 9 for more information. Readme file is present in both describing how to build and run the software.

## 5 Part II: Replay attack

During the analysis we further examined some parts of the code in an attempt to discover potential vulnerabilities to exploit. Let us now briefly describe how Telegram processes all incoming data.

### 5.1 Incoming data processing

The Figure 8 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 4.

Next, the ConnectionsManager::onConnectionDataReceived() function is called. If the auth_key_id is not set to 0 (which is only the case during the key exchange process), 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 8: 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.](image)

### 5.2 The Vulnerability

One of the most notorious ways of breaching a messenger’s security is what is called a replay attack. We analyzed how Telegram deals with this issue, see Listing 3. 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.

The behavior we believe might be exploitable concerns the addProcessedMessageId() function. The function checks the size of the array and if it exceeds 300, it erases the first 100 messages as seen in Listing 4.

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

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

Listing 3: 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.

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

Listing 4: 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.

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.

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. This method proved to be unfruitful so we instead 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.

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.

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 [16, 12].

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” [9]. 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 [10].

The VM installs all the required software and Trudy itself as well. Then it routes all the traffic in and out of Trudy using iptables. Trudy receives all the traffic, modifies it based on the used module and sends it back to internet as seen in Figure 9.

![Figure 9: The computer runs Trudy inside a virtual machine. All traffic is routed bidirectionally through Trudy.](image)

5.3.1 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”

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

We implemented the Trudy module as shown in Listing 5. The module 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 testing device 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 module further checks whether some bytes were already saved. If not, it signifies this is the first message and it copies the TCP payload into an 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.

```go
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 5: 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.

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 did not continue to pursue this route, however, we believe this attack is feasible. In order for it to work 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

6 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 [4]. 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 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 [14].
Among other checks, the Security Guidelines require [14] 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[2]. 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 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 and fixed this issue in the next Android update – version 3.16 – which was released in early January 2017. The developers modified the addProcessedMessageId() and more importantly the isMessageIdProcessed() functions.

The addProcessedMessageId() function now saves minProcessedMessageId value. This variable is set when the array of message IDs exceeds its 300 items capacity. It is set to the current lowest message ID stored in the array. All processed IDs have lower ID than this value and the check correctly asserts that. We’ve reviewed the code and concluded its correctness.

7 Conclusion

In the scope of this article we have firstly introduced the MTProto protocol and Telegram itself. We have documented the protocol’s internal working, mainly the encryption process and we have also noted the cryptographic primitives it uses.

Furthermore, we have described the undocumented obfuscation method. 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 removed with the deobfuscation program we have implemented. We have reported our findings to the Telegram security team, which noted that the method is used to circumvent some of the less sophisticated methods of censorship in certain countries.

The rival Signal messenger recently incorporated a mechanism to circumvent such censorship. The technique called domain fronting uses popular cloud services like the ones from Google or Amazon as a middle man to route all traffic. More information are available on the Signal creators’ blog [11] or in the original paper [5]. We believe Telegram creators should take inspiration from such solutions to create more sophisticated mechanisms to avoid censorship.

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 applicable, 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 fixed this issue in the following software release.

Our work mainly focused 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.

8 Acknowledgments

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. And finally, special thanks to Miriam for proofreading.

9 Availability

All tools are available on the author’s public GitHub profile (github.com/tsusanka). In particular, the Telegram Deobfuscator, as described in Section 4.2, is available at https://github.com/tsusanka/telegram-deobfuscator and the scripts to extract secrets from the Android application are available at https://github.com/tsusanka/telegram-extractor.
References

[1] AHMED, M. Telegram hits 100m users and commits to remaining ad-free [online], Feb. 2016. [accessed 2016-12-16].


[8] LEYDEN, J. Homebrew crypto in Telegram hangout app full of holes, say security pros [online], Nov. 2015. [accessed 2016-12-16].


[16] TELERIK. Fiddler homepage [online]. [accessed 2016-12-03].

Notes

1Telegram actually uses BoringSSL – a fork of OpenSSL by Google, but for our case the distinction was not relevant.

2According to Telegram, this is because of the checks done in the MessagesController.java class on lines 5731, 5561, 5765, 5817 and others.