The Cryptanalysis of Telegram

December 16, 2016

Submitted by:

Marvin Liu, Eddie Smith, Kushmitha Unnikumar

Professor:

Dr. Nadia Heninger

December 16, 2016

Computer and Information Science

University of Pennsylvania

1 Abstract

In the modern age, secrecy of conversation is a highly sought-after quality. There are many providers who claim to have this service, and one of these providers is Telegram. However, the approaches to conversation secrecy are varied, leading to differences in protocol security and possibly being insecure. We attempt to provide an analysis of whether or not the protocol used by Telegram, MTProto, is substantially secure to recommend as a secret conversation client. We examine the various structures in their protocol and elaborate known vulnerabilities for certain pieces of the protocol. After this elaboration, we piece together a framework for a possible attack on MTProto. This protocol is lacking in security in several areas, and we find that it is not a protocol that we would recommend for use.

2 Introduction

Many people wish to keep what they send to their contacts secure, whether it is for corporate purposes or even for day-to-day uses. There are many phone applications that incorporate sending encrypted data that claim they are the most "secure" to use. To begin the first step of preserving the safety and privacy of these types of users, we will be doing a cryptanalysis of Telegram.

2.1 Telegram

Telegram is a cloud-based mobile and desktop messaging app with a focus on security and speed. People are able to use this app on all registered devices at the same time while achieving fast and reliable messaging. Telegram also touts itself to use end-to-end encryption, or device specific secret chats [10]. However, the issue with this that this option of end-to-end encryption is not its default, which causes many people to believe that they are sending secret chats when they actually are not. Another issue with Telegram is that it has created its own security protocol, MTProto, and as many people familiar with the cryptographic world, there is a saying: never make your own crypto.

2.2 MTProto

MTProto is a custom symmetric encryption scheme based on a 256-bit symmetric AES encrytion in IGE mode, RSA 2048 encryption, and Diffie-Hellman key exchange. It is made up of four primary components: a device registration protocol, key exchange for end-to-end encrypted secret chat, message encryption, and key derivation function (KDF).

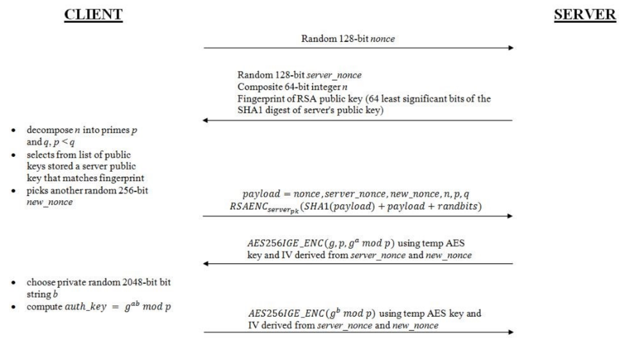

CLIENT
SERVER
Random 128-bit *nonce*
Random 128-bit *server_nonce*
Composite 64-bit integer n
Fingerprint of RSA public key (64 least significant bits of the SHA1 digest of server's public key)
decompose n into primes p and q, $p < q$
selects from list of public keys stored a server public key that matches fingerprint
picks another random 256-bit *new_nonce*
$payload = nonce, server\_nonce, new\_nonce, n, p, q$
$RSAENC_{server_{pk}}(SHA1(payload) + payload + randbits)$
$AES256IGE\_ENC(g, p, g^a \bmod p)$ using temp AES key and IV derived from *server_nonce* and *new_nonce*
choose private random 2048-bit bit string b
compute $auth\_key = g^{ab} \bmod p$
$AES256IGE\_ENC(g^b \bmod p)$ using temp AES key and IV derived from *server_nonce* and *new_nonce*

Figure 1: Diffie Hellman Key Exchange (Device Registration)

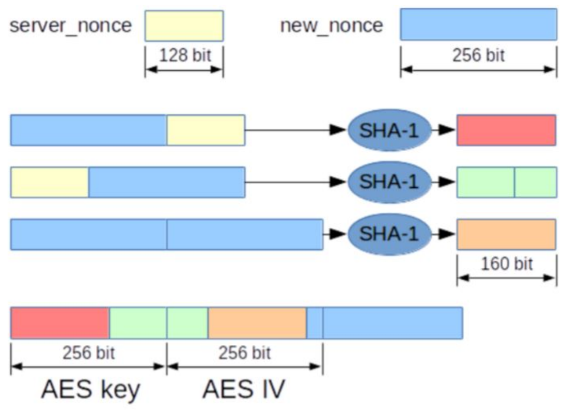


Figure 2: AES key and IV derivation from the *server_ nonce* and *new_ nonce*

2.2.1 Device Registration Protocol

The device registration protocol is based on the Diffie-Hellman Key Exchange protocol as shown in Figure 1. First, the client will generate a random 128-bit bit string *nonce*, and the server will respond by sending its own random 128-bit bit string *server_ nonce*, a composite 64-bit integer n, and the fingerprint of an RSA public key. The fingerprint is simply the 64 least significant bits of the SHA-1 digest of the server's public key. Once the client receives this information, he will first decompose n into primes p and q such that $p < q$. Then he will pick a server public key $server_{pk}$ that matches the RSA fingerprint from a list of public keys that is already available to him. Finally, he will choose a random 256-bit bit string *new_ nonce*. When all of these values are obtained, the client will then send over the payload, which consists of the *nonce*, *server_ nonce*, *new_ nonce*, n, p, and q, and the RSA encrypted value of $SHA1(payload)||payload||(any\ random\ bytes)$ by using the $server_{pk}$ such that adding the random bytes will make the length equivalent to 255 bytes.

The server will respond by sending an AES-256-IGE encrypted value of g, p, $g^a \mod p$ by using a temporary AES key and IV derived from the *server_ nonce* and the *new_ nonce*, respectively. The client will then choose a private random 2048-bit bit string b, computes $g^{ab} \mod p$ as the authentication key *auth_ key*, and finally sends an AES-256-IGE encrypted value of $g^b \mod p$ by using the temporary AES key and IV derived from the *server_ nonce* and the *new_ nonce*, respectively [6, 10].

The way the temporary AES key and IV are derived from the *server_ nonce* and the *new_ nonce* is illustrated in Figure 2. Basically, three SHA-1 values are computed with different combinations of the server_nonce and new_nonce. These SHA-1 outputs, along with a part of the *new_ nonce*, are then used to generate the key and IV. More specifically, the AES key is equivalent to:

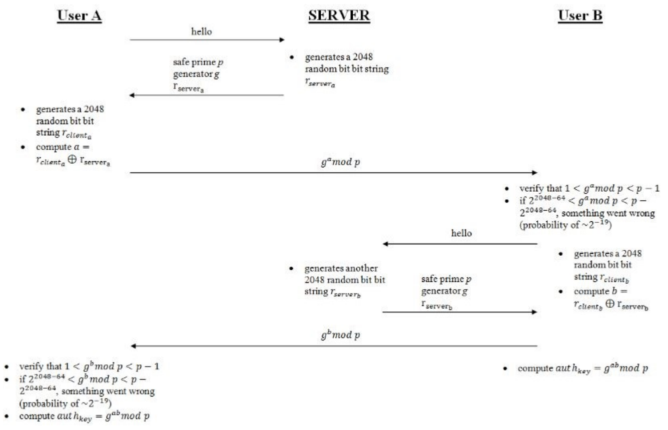
$SHA1(new\_nonce||server\_nonce)||substr(SHA1(server\_nonce||new\_nonce), 0, 12)$

and the AES IV is equivalent to:

$substr(SHA1(server\_nonce||new\_nonce), 12, 8)||SHA1(new\_nonce||new\_nonce)||substr(new\_nonce, 0, 4)$

[6, 10].

Now, the device with the application is ready to communicate with another authorized user through a regular chat.



User A
SERVER
User B
hello
generates a 2048 random bit bit string r_{server_a}
safe prime p generator g r_{server_a}
generates a 2048 random bit bit string r_{client_a}
compute $a = r_{client_a} \oplus r_{server_a}$
$g^a mod\ p$
verify that $1 < g^a mod\ p < p - 1$
if $2^{2048-64} < g^a mod\ p < p - 2^{2048-64}$, something went wrong (probability of $\sim 2^{-19}$)
hello
generates a 2048 random bit bit string r_{client_b}
compute $b = r_{client_b} \oplus r_{server_b}$
generates another 2048 random bit bit string r_{server_b}
safe prime p generator g r_{server_b}
$g^b mod\ p$
verify that $1 < g^b mod\ p < p - 1$
if $2^{2048-64} < g^b mod\ p < p - 2^{2048-64}$, something went wrong (probability of $\sim 2^{-19}$)
compute $auth_{key} = g^{ab} mod\ p$
compute $auth_{key} = g^{ab} mod\ p$

Figure 3: Diffie Hellman Key Exchange (Key Exchange)

2.2.2 Key Exchange

Again, the key exchange protocol is based on the Diffie-Hellman Key Exchange protocol, as shown in Figure 3. Let us say that Alice (User A) and Bob (User B) are trying to initiate a secret chat with each other. First, Alice will start communications with the server and the server will respond by generating a random 2048-bit bit string r_{server_a}. The server will then send over a safe prime p such that $q = \frac{p-1}{2}$ is prime and $2^{2047} < p < 2^{2048}$, generator g that is equal to a number between 2 and 7 such that g generates a cyclic subgroup of order q, and r_{server_a}. Once Alice obtains these values, she will generate her own random 2048-bit bit string r_{client_a} and compute $a = r_{client_a} \oplus r_{server_a}$. She will then send $g^a \mod p$ to Bob. Bob will then verify that $1 < g^a \mod p < p - 1$. He will also check to see if $2^{2048-64} < g^a \mod p < p - 2^{2048-64}$, and if it does, then something must have gone wrong after Alice sent over this value since it would only occur with a probability of 2^{-19}. If these conditions are satisfied and $g^a \mod p$ appears to not be tampered with, Bob will start communications with the server. The server will respond as before; it will generate a random 2048-bit bit string r_{server_b} and send over the same safe prime p, the same generator g, and r_{server_b}. Bob will follow what Alice did before and generate his own random 2048-bit bit string r_{client_b} and compute $b = r_{client_b} \oplus r_{server_b}$ so that he can send over $g^b \mod p$ to Alice. Alice will then follow the same verification process that Bob went through, and if $g^b \mod p$ passes both conditions, she and Bob can now compute $g^{ab} \mod p$ as the authentication key $auth\_key$ [6, 10].

Now, two authorized users are ready to communicate with one another through a secret chat.

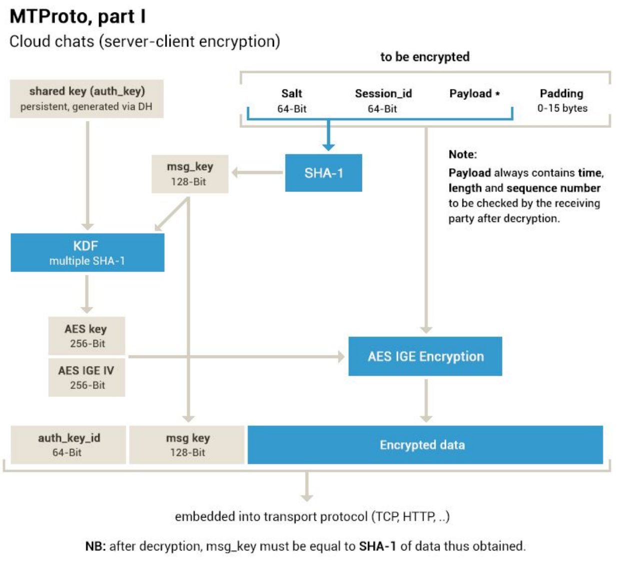

MTProto, part I
Cloud chats (server-client encryption)
to be encrypted
shared key (auth_key)
persistent, generated via DH
Salt
64-Bit
Session_id
64-Bit
Payload *
Padding
0-15 bytes
msg_key
128-Bit
SHA-1
Note:
Payload always contains time, length and sequence number to be checked by the receiving party after decryption.
KDF
multiple SHA-1
AES key
256-Bit
AES IGE IV
256-Bit
AES IGE Encryption
auth_key_id
64-Bit
msg key
128-Bit
Encrypted data
embedded into transport protocol (TCP, HTTP, ..)
NB: after decryption, msg_key must be equal to SHA-1 of data thus obtained.

Figure 4: Server-Client encryption scheme

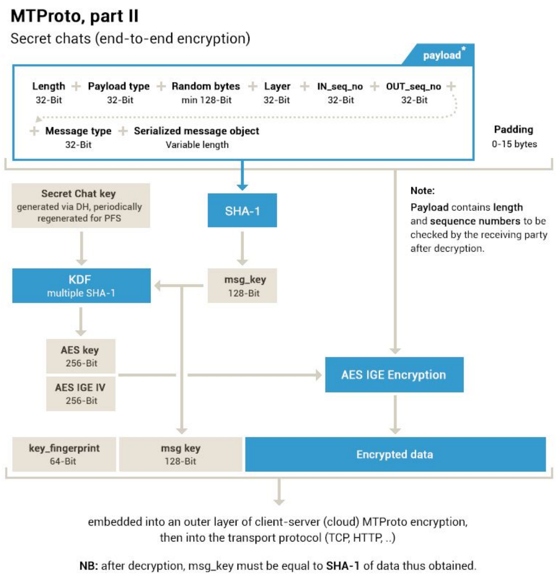
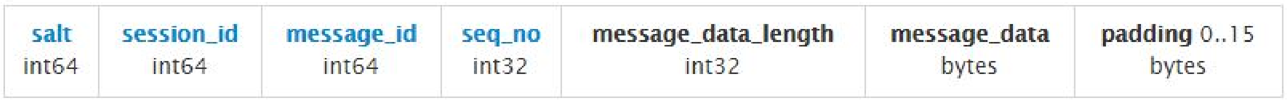

MTProto, part II
Secret chats (end-to-end encryption)
payload*
Length
32-Bit
Payload type
32-Bit
Random bytes
min 128-Bit
Layer
32-Bit
IN_seq_no
32-Bit
OUT_seq_no
32-Bit
Message type
32-Bit
Serialized message object
Variable length
Padding
0-15 bytes
Secret Chat key
generated via DH, periodically regenerated for PFS
SHA-1
Note:
Payload contains length and sequence numbers to be checked by the receiving party after decryption.
KDF
multiple SHA-1
msg_key
128-Bit
AES key
256-Bit
AES IGE IV
256-Bit
AES IGE Encryption
key_fingerprint
64-Bit
msg key
128-Bit
Encrypted data
embedded into an outer layer of client-server (cloud) MTProto encryption, then into the transport protocol (TCP, HTTP, ..)
NB: after decryption, msg_key must be equal to SHA-1 of data thus obtained.

Figure 5: End-to-end encryption scheme

2.2.3 Message Encryption

The message encryption scheme for regular chats is shown in Figure 4. The shared key ($auth\_key$) is generated from the Diffie Hellman Key Exchange described in 2.2.2 and is only generated once for all messages sent from one user to another. The message key msg_key is the lower-order 128 bits of SHA1($salt||session\_id||payload$). The $salt$ is a random 64-bit number that is periodically generated at the request of the server such that all subsequent messages contain this salt. The $session\_id$ is a random 64-bit number that is generated by the client so that individual sessions, such as different instances of the application that are created with the same auth_key, can be distinguished. The $payload$ contains a 24 byte header, which consists of a 64-bit key identifier and a 128-bit message key, concatenated with the encrypted version of the plaintext message, which consists of a time-dependent 64-bit number, sequence number, length, and the message data. A visualization of the data to be encrypted is shown in Figure 6. The $auth\_key$ and msg_key are then passed into a KDF that is described in more detail in 2.2.4, resulting in an AES key and AES IGE IV being output so that they can be used in the AES-IGE encryption of the data from Figure 6. Finally, the value of $auth\_key\_id||msg\_key||encrypted\_data$ is the encrypted message that is sent over, where $auth\_key\_id$ is the lower-order 64 bits of the SHA1($auth\_key$) [10].



salt	session_id	message_id	seq_no	message_data_length	message_data	padding 0..15
int64	int64	int64	int32	int32	bytes	bytes

Figure 6: Payload

The end-to-end encryption scheme for secret chats is shown in Figure 5. This encryption scheme works the same way as the one for regular chats except for a few differences. The *payload* in this case is made up of a different combination of values. The secret chat key is generated via Diffie-Hellman like the shared key in the regular chat, however, it is only periodically regenerated so that perfect forward secrecy is satisfied. The key_fingerprint is similar to the *auth_key_id* and is the lower-order 64 bits of the SHA1(secret chat key) [10].

2.2.4 Key Derivation Function

The KDF is shown in Figure 7 and is shown being used in Figure 4 and Figure 5. It takes the inputs of *auth_key* and *msg_key* and calculates the following, where $x = 0$, the numerical values are in bytes, and the *substr()* function is of the form substr(*string*, *starting_byte_value*, number of bytes of the *string* from the *starting_byte_value*) [6, 10]:

- $sha1\_a$ = SHA1(msg_key + substr($auth\_key$, x, 32));
- $sha1\_b$ = SHA1(substr($auth\_key$, 32 + x, 16) + msg_key + substr($auth\_key$, 48 + x, 16));
- $sha1\_c$ = SHA1(substr($auth\_key$, 64 + x, 32) + msg_key);
- $sha1\_d$ = SHA1(msg_key + substr($auth\_key$, 96 + x, 32));
- aes_key = substr($sha1\_a$, 0, 8) + substr($sha1\_b$, 8, 12) + substr($sha1\_c$, 4, 12);
- aes_iv = substr($sha1\_a$, 8, 12) + substr($sha1\_b$, 0, 8) + substr($sha1\_c$, 16, 4) + substr($sha1\_d$, 0, 8);

The *aes_key* and *aes_iv* are then output and used for AES-256-IGE encryption.

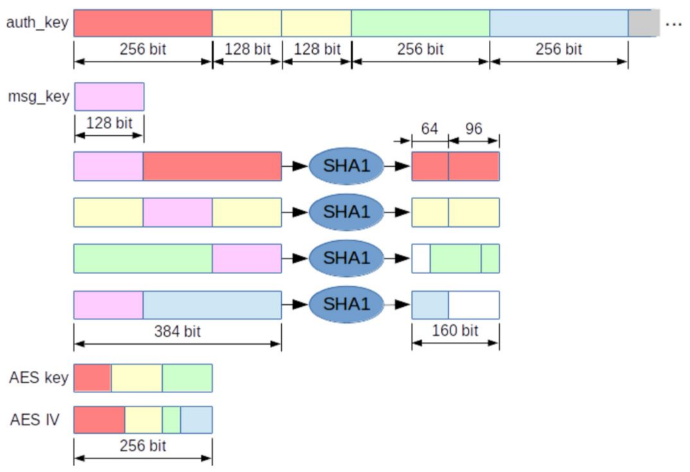


Figure 7: Key Derivation Function

3 Known Partial Weaknesses

3.1 AES IGE

Telegram uses AES-IGE which is a mode of operation for block ciphers with symmetric key encryption. The ith cipher block is obtained by $c_i = f_k(m_i \oplus c_i - 1) \oplus m_i - 1$ where f_k is the underlying block cipher encryption function with key k. As shown in Figure 8, the encryption requires two initialization vectors C_0 and m_0. While m_0 can be the first message block, C_0 could be either an arbitrary block given as a parameter or be assigned as $f_{k_0}(m_0)$ where k_0 is a second random key [7].

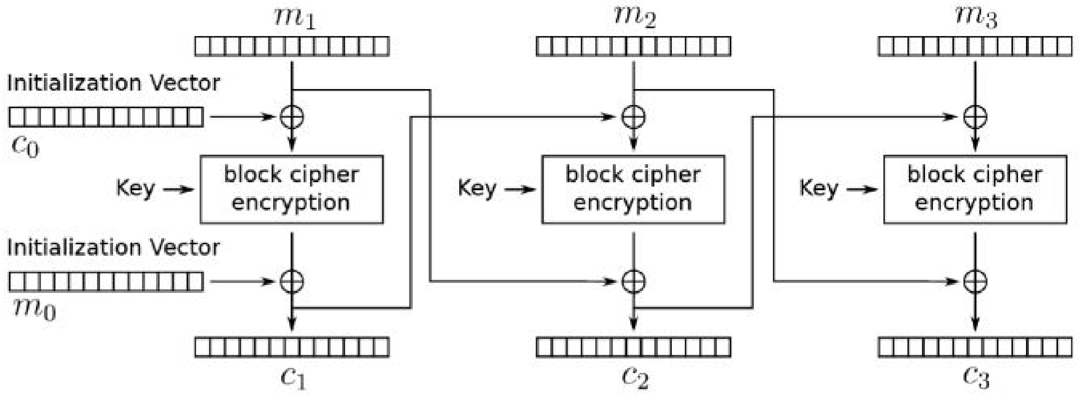


Figure 8: AES-IGE encryption

The decryption is carried out by computing $m_i = c_{i-1} \oplus f_k(c_i \oplus c_{i-1})$. As can be seen in Figure 9, this mode of operation is non-malleable since flipping one bit in a ciphertext block can cause the decrypted block as well as all the other blocks to get garbled. Thus, the errors are propagated forward indefinitely [7].

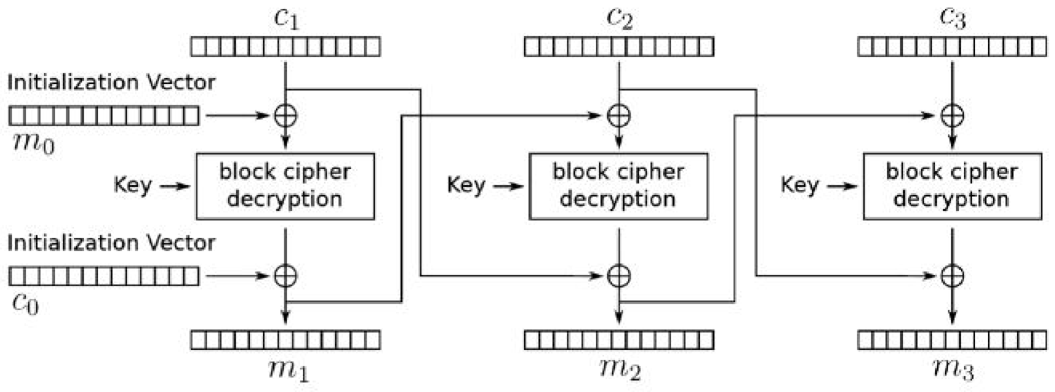
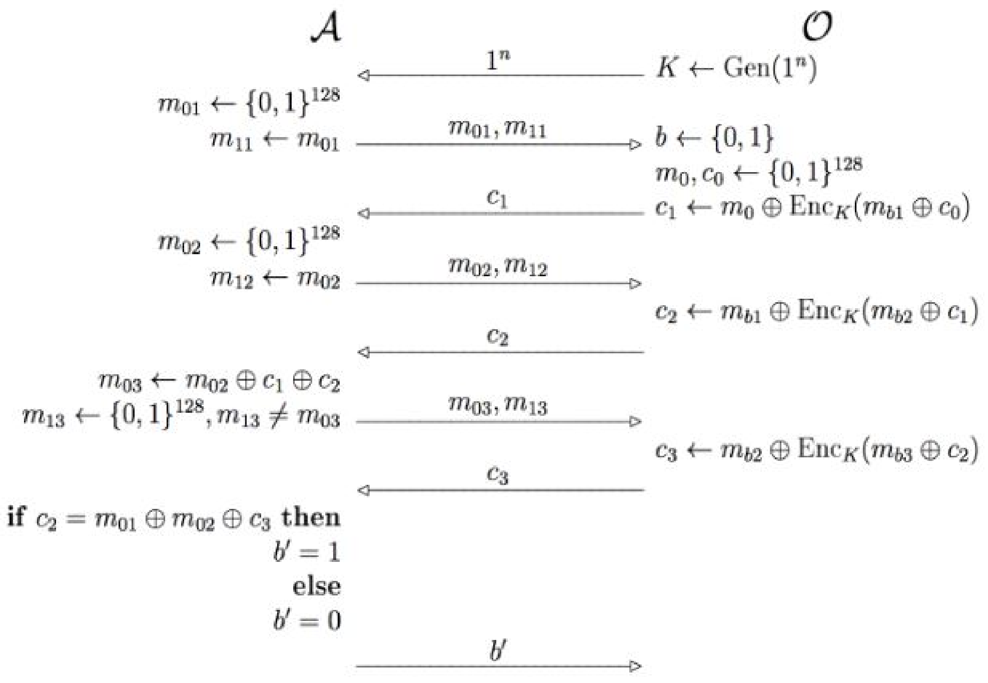


Figure 9: AES-IGE decryption

While AES-IGE is IND-CPA secure, it has been proven that it is insecure to a Blockwise-adaptive chosen plaintext attack (BACPA). BACPA allows the attacker decide upon the next block after viewing the results of inserting one or more blocks of choice into a plaintext message [2]. The adversary can win the security game against AES-IGE encryption if he finds a collision between two blocks before they enter the encryption function and after they are XORed with the ciphertexts. An example of this attack is described in Figure 10.

The reason for choosing m_{03} the way it is is because if $b = 0$, then $m_{01} \oplus m_{02} \oplus c_3 = m_{01} \oplus m_{02} \oplus m_{02} \oplus Enc_k(m_{02} \oplus c_1) = m_{01} \oplus Enc_k(m_{02} \oplus c_1) = c_2$. This will allow A to succeed with probability 1.



$\mathcal{A}$
$\mathcal{O}$
1^n
$K \leftarrow \text{Gen}(1^n)$
$m_{01} \leftarrow \{0,1\}^{128}$
$m_{11} \leftarrow m_{01}$
m_{01}, m_{11}
$b \leftarrow \{0,1\}$
$m_0, c_0 \leftarrow \{0,1\}^{128}$
c_1
$c_1 \leftarrow m_0 \oplus \text{Enc}_K(m_{b1} \oplus c_0)$
$m_{02} \leftarrow \{0,1\}^{128}$
$m_{12} \leftarrow m_{02}$
m_{02}, m_{12}
$c_2 \leftarrow m_{b1} \oplus \text{Enc}_K(m_{b2} \oplus c_1)$
c_2
$m_{03} \leftarrow m_{02} \oplus c_1 \oplus c_2$
$m_{13} \leftarrow \{0,1\}^{128}, m_{13} \neq m_{03}$
m_{03}, m_{13}
$c_3 \leftarrow m_{b2} \oplus \text{Enc}_K(m_{b3} \oplus c_2)$
c_3
if $c_2 = m_{01} \oplus m_{02} \oplus c_3$ **then**
$b' = 1$
else
$b' = 0$
b'

Figure 10: IND BACPA attack

3.2 SHA-1

MTProto's Key Derivation Function and message key generation both rely on the SHA-1 hashing function. The message key is derived from the lower 128 bits of the SHA-1 hash of the payload as shown in Figure 4 and Figure 5. The Key Derivation Function uses four instances of SHA-1, using both full and partial results (Figure 7). The use of SHA-1 has been deprecated for several years [3], and the vulnerability of the function has only grown since then. There have been near-collision attacks in under 2^{60} compressions [8] and more recently a full freestart collision in practical time [9]. Despite this, Telegram continues to use SHA-1 over a SHA-2 or SHA-3 family function with a justification of an improvement in performance [10]. SHA-1 was also previously used in Telegram's secret chat verification, leaving it vulnerable to a Man in the Middle attack for DH key generation. This functionality has since been replaced with a combination of SHA-1 and SHA-256.

3.3 Man-in-the-middle Attacks

3.3.1 Malicious Server MitM attack

Because the protocol uses Diffie-Hellman for key generation, it requires a level of trust between the users and the server itself. This allows for the possibility of a malicious server man-in-the-middle attack to occur. Although this is only possible under the assumption that users will send their public Diffie-Hellman value to the server before it sends them the nonce. This vulnerability is shown in Figure 11 but is a part of an older version of Telegram. Now, both of the nonces that is provided by the server results in the final keys to be identical. However, the shared secret of g_{AB} is now easy to obtain because $g_{AB} = g_{AS} \oplus A_{nonce} = g_{AS} \oplus g_{AS} \oplus g_{AB} = g_{AB}$. Even though Telegram had since gotten rid of this, it leads to many people thinking that there is an intentional backdoor that Telegram has built so they can access every user's end-to-end encrypted messages [6].

3.3.2 Naive Third Party MitM attack

As specified by Telegram's protocols [10], they provide users with key visualizations so that they can confirm that no man-in-the-middle attacks have taken place. This is how Telegram guarantees authenticity. This

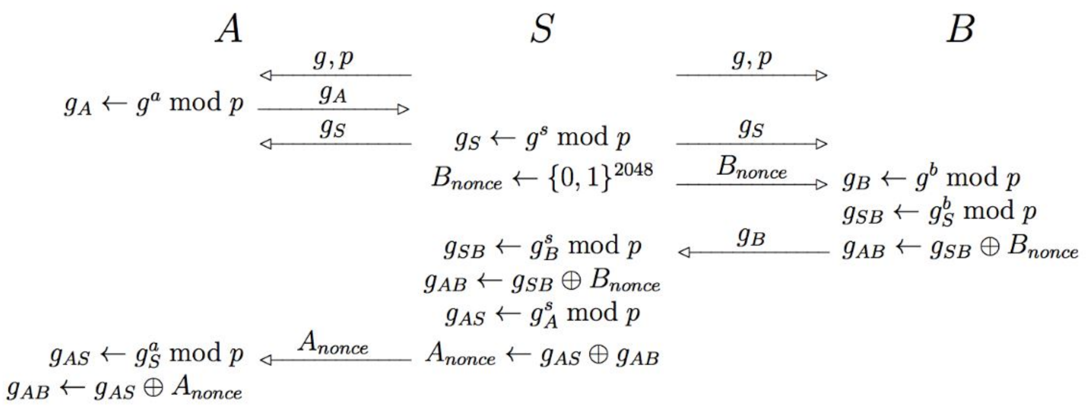

A
S
B
g, p
g, p
$g_A \leftarrow g^a \bmod p$
g_A
g_S
$g_S \leftarrow g^s \bmod p$
g_S
$B_{nonce} \leftarrow \{0,1\}^{2048}$
B_{nonce}
$g_B \leftarrow g^b \bmod p$
$g_{SB} \leftarrow g_S^b \bmod p$
$g_{SB} \leftarrow g_B^s \bmod p$
g_B
$g_{AB} \leftarrow g_{SB} \oplus B_{nonce}$
$g_{AB} \leftarrow g_{SB} \oplus B_{nonce}$
$g_{AS} \leftarrow g_A^s \bmod p$
$g_{AS} \leftarrow g_S^a \bmod p$
A_{nonce}
$A_{nonce} \leftarrow g_{AS} \oplus g_{AB}$
$g_{AB} \leftarrow g_{AS} \oplus A_{nonce}$

Figure 11: Malicious Server MitM Attack

visualization/fingerprint is created from using the first 128 bits of SHA1(initial key) followed by the first 160 bits of SHA256(key used when secret chat was updated to layer 46) and looks similar to what is shown in Figure 12. Because users need to meet in person to actually compare these visualizations, it is easy to exploit this and conduct a naive third party man-in-the-middle attack as shown in Figure 13. Meeting in person usually defeats the purpose of using a chatting service, which is why this attack is still feasible to do. Many people will usually screenshot the visualization and send it over a newly instantiated, unencrypted, unauthenticated chat, which can be exploited by having the eavesdropper replace this screenshot with one of his own. However, this attack will only work if the users do not physically meet because if they do, they will be able to see that the visualizations do not match.

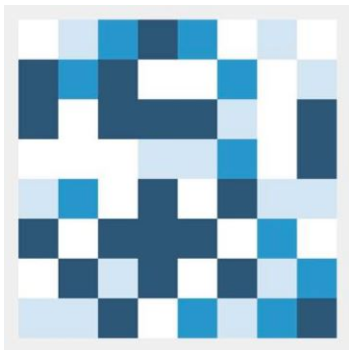


Figure 12: Key Visualization

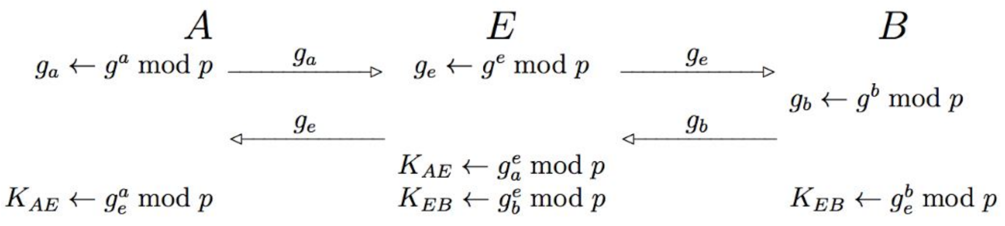

A
E
B
$g_a \leftarrow g^a \bmod p$
g_a
$g_e \leftarrow g^e \bmod p$
g_e
$g_b \leftarrow g^b \bmod p$
g_e
g_b
$K_{AE} \leftarrow g_a^e \bmod p$
$K_{AE} \leftarrow g_e^a \bmod p$
$K_{EB} \leftarrow g_b^e \bmod p$
$K_{EB} \leftarrow g_e^b \bmod p$

Figure 13: Naive Third Party MitM Attack

3.4 Padding Validation

Message authentication in MTProto is conducted using a verification of the message key. The encrypted data is decrypted and the lower 128 bits of the SHA-1 of the decrypted payload is compared to the transmitted message key. If the two are identical, then the message is successfully authenticated. A visualization of the decryption and authentication process is presented in Figure 14.

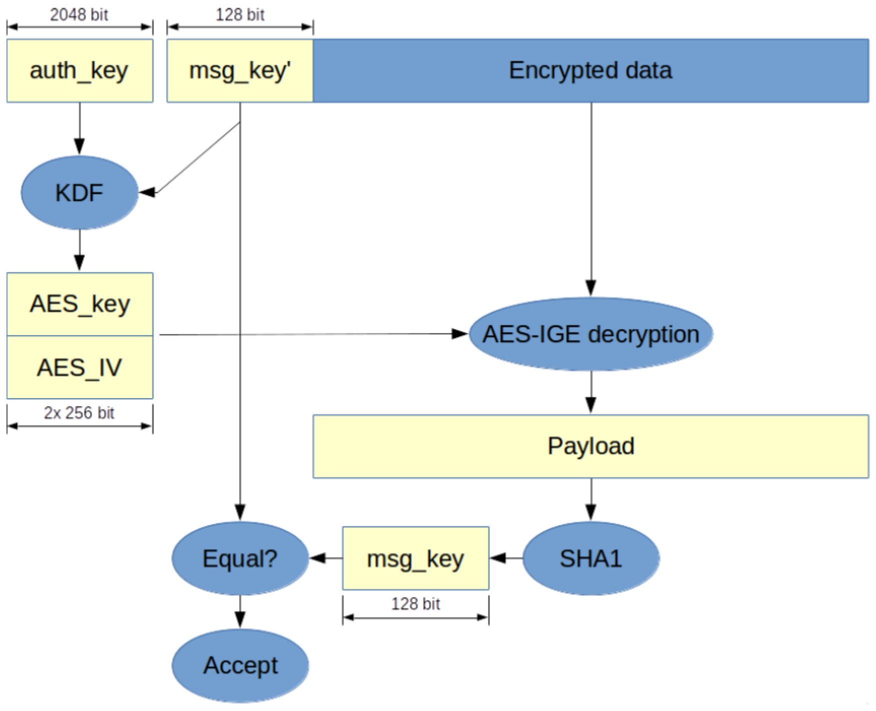


Figure 14: Message Authentication and Decryption

Provided that padding is done properly, there would be no issues with this scheme for validation. However, as one can see in Figures 4 and 5, the padding is not added until after computation of the message key, which means that the padding is vulnerable to manipulation. As described by Jakobsen [5], this leads to two separate chosen ciphertext attacks on MTProto.

3.4.1 Padding Length Extension

The payload contains as a part of its content the length of the message, which means that the length of the padding is insignificant so long as it decrypts properly. Investigation of the padding in MTProto will show that this padding is rather rather than conforming to a standard like PKCS. An adversary under a chosen-ciphertext attack can therefore add to the padding as they wish, since it will simply be thrown away. This ciphertext is distinct from the original one received, so the attacker wins with a probability of 1. More formally,

1. $\mathcal{A}$ outputs M_0, M_1 s.t. $M_0 \neq M_1$, and M_0 and M_1 have the same length
2. $\mathcal{O}$ chooses $b \in \{0, 1\}$ and sends $C_b = Enc_k(M_b)$
3. $\mathcal{A}$ appends padding to C, making $C' \neq C$, asks for decryption
4. $\mathcal{O}$ outputs $M' = Dec_k(C')$ after reading payload length
5. Either $M' = M_0$ or $M' = M_1$, so $\mathcal{A}$ knows the value of b

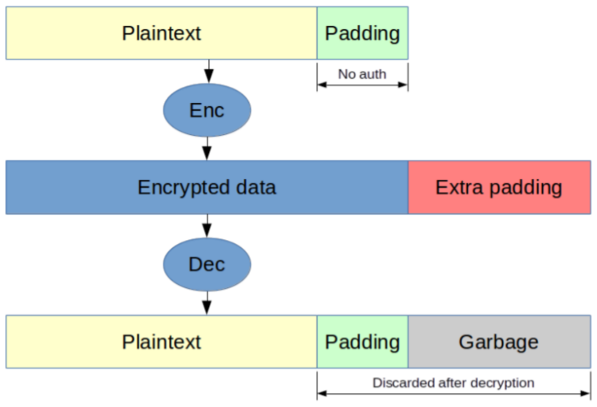


Figure 15: Padding Length Extension Attack

3.4.2 Padding Plaintext Collision

In addition to adding to the length of the padding, one can also modify the existing padding to have a non-negligible chance of colliding in the plaintext. We revise the CCA game as follows:

1. $\mathcal{A}$ sends M, length equal to $1 \mod 16$ bytes
2. MTProto hashes M into message key using SHA-1
3. Prior to encryption, 15 random bytes of padding are appended, then ciphertext $C = Enc_k(M)$ is generated
4. $\mathcal{A}$ modifies last block to get $C' \neq C$
5. $\mathcal{A}$ outputs C'

On decryption of C' to M', the last block of M' will be garbled due to the workings of IGE. However, in computing the SHA-1 to generate the message key, we only require that the first bit of this block be the same as it was in M. The chances of doing so are 2^{-8}. This would be ideal, but examination of MTProto yields that all lengths be made into multiples of length 4 bytes, with any unused bytes made into zeroes [10]. This means that we cannot have a message of length 1 mod 16 bytes. Our best scenario is then where the length of the message is 4 mod 16 bytes. Generating a collision over four bytes has a probability of 2^{-32}. While much less feasible than our collision with probability 2^{-8}, this is certainly within the capabilities of a resourceful attacker.

4 Analysis

4.1 IND-CCA security

While Telegram's susceptibility to a theoretical IND-CCA attack is a major loophole, it has not been considered as a real-life vulnerability yet. An adversary can modify a ciphertext so that the oracle will accept it and decrypt to the same plaintext as the original unmodified ciphertext, but the contents of the ciphertext are known only to the sender and the receiver, thus giving no advantage to the adversary. There has been no attack on Telegram yet that has resulted in a full plaintext recovery [10].

4.2 Forward Secrecy and Diffie-Hellman

It should be noted first and foremost that the entire security of the system relies on the security of DH key exchange and subsequent AES key derivation. The message key is published along with the encrypted text, and a SHA-1 partial collision can be calculated to find other plaintexts that produce the same message key. This leaves the protocol open to exploitation through persistent messaging to discover information about the AES key. To prevent replay attacks and limit such activity, Telegram includes the time as part of their payload, and any message is rejected over 300 seconds after it is created or 30 seconds before it is created [10]. For now, this places severe restrictions on the plaintexts that the adversary can have encrypted. The adversary becomes limited by their ability to cause SHA-1 collisions quickly, though this will continue to improve with time.

It should also be noted that perfect forward secrecy exists only in the context of Telegram's secret chats which periodically produce a new secret chat key via Diffie-Hellman. This is not done in the standard client-server communication, which is the mode enabled by default. The authentication key is generated only once, so anyone with access to both the authentication key for a user and the message keys (which are transmitted as part of the data) can gain access to all messages sent by that user. For secret chats, the key is regenerated every 100 messages or one week [10], meaning that the scope of conversation gained from accessing one key is somewhat limited.

4.3 Message Authentication

Another major issue with MTProto is that it lacks a strong sense of message authentication. Instead of using a true MAC, MTProto opts to use a simple SHA-1 of the plaintext. This (termed the message key by the protocol itself) is transmitted along with the encrypted text. This is close to an Encrypt-and-MAC scheme, which provides neither CPA nor CCA security [4]. The use of SHA-1 over a true MAC makes this even more vulnerable. Telegram states that a more secure Encrypt-then-MAC scheme is not necessary since the plaintext contains information not known to the attacker such as message length and session ID [10]. However, a SHA-1 partial collision can be used to generate the same message key with a different plaintext in a feasible amount of time as discussed earlier.

4.4 Device Attacks

All of this assumes that an adversary is going to try and attack Telegram's encryption in the middle. In reality, an adversary is just as likely to try and exploit other avenues. It is possible to orchestrate an active attack on a device to gain control of it and view both encrypted and unencrypted secret chat messages. After all, the messages must be encrypted and stored somewhere, since they are not sent to the server. Using a kernel exploit and a process memory dump, one can gain access to the unencrypted messages sent and received by a device using Telegram's secret chats [1]. Thus, one does not only have to worry about the security of the encryption system but also of the application as a whole.

5 Conclusion

Our cryptanalysis of Telegram shows that it has many vulnerabilities that must be fixed before becoming truly end-to-end encrypted. Since our cryptanalysis is all theoretical, in the future, we would like to implement an actual attack on MTProto by following the guidelines as provided by Telegram [10]. By doing so, we can show that MTProto is as insecure as we have analyzed it to be and build or recommend a better system and protocol that Telegram can use. Although we are not professional cryptographers, we could still provide these vulnerabilities to Telegram and discuss why they should modify their protocol to be more secure than it currently is. We hope to show that Telegram is not secure for secret chats and lead them to construct a more secure protocol in the near future. Therefore, we would not recommend using Telegram as an end-to-end encrypted chat service.

References

[1] Zuk Avraham. Telegram app store secret-chat messages in plain-text database. Internet: https://blog.zimperium.com/telegram-hack/, 2016. Retrieved December 16, 2016, from https://blog.zimperium.com/telegram-hack/.

[2] Gregory V Bard. Modes of encryption secure against blockwise-adaptive chosen-plaintext attack. *IACR Cryptology ePrint Archive*, 2006:271, 2006.

[3] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 800:131A, 2011.

[4] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 531–545. Springer, 2000.

[5] Jakob Jakobsen and Claudio Orlandi. On the cca (in) security of mtproto.

[6] Jakob Jakobsen and Claudio Orlandi. *A practical cryptanalysis of the Telegram messaging protocol.* PhD thesis, Master Thesis, Aarhus University (Available on request), 2015.

[7] mgp25. Aes ige encryption. Internet: https://www.mgp25.com/AESIGE/, 2016. Retrieved December 16, 2016, from https://www.mgp25.com/AESIGE/.

[8] Marc Stevens. New collision attacks on sha-1 based on optimal joint local-collision analysis. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 245–261. Springer, 2013.

[9] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full sha-1. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 459–483. Springer, 2016.

[10] Telegram. Telegram messenger. Internet: https://telegram.org, 2016. Retrieved December 16, 2016, from https://telegram.org.