Seguridad en Aplicación de Mensajería Telegram

Jesús Márquez, Lucas Salazar, Pablo Yañez

***Abstract*—La utilización de encriptación y funciones hash ha llevado el concepto de protección a diferentes protocolos que aseguran autenticación, integridad y confidencialidad o cubren un par de estas. A continuación se presentará la aplicación Telegram donde se centrará el desarrollo sobre su seguridad y cómo ésta es implementada. Telegram utiliza un protocolo llamado MTProto que posee encriptación AES-IGE (basada en CBC), función hash SHA-256 utilizada como MAC y derivación de claves (KDF) basado en SHA-256, entre otras cosas, de tal manera de brindar autenticación, integridad y confidencialidad dejando a esta aplicación protegida contra espías, terceros que inserten mensajes falsos, suplantación de Id, secuestros de sesión y denegación de servicio.**

I. Introducción

En la actualidad, dada la gran densidad de celulares inteligentes presente en la sociedad, ha prosperado el servicio de mensajería instantánea como medio de comunicación. Es por ello que la privacidad y seguridad de estos servicios son de gran importancia; una de las aplicaciones que fomenta dichas propiedades es Telegram.

Telegram, como se menciona anteriormente, es un servicio de mensajería instantánea que se caracteriza por tener una gran variedad de implementaciones de clientes, por lo que es posible correr la aplicación desde celulares inteligentes, navegadores web y aplicaciones de escritorios, por mencionar algunos. Pero en definitiva, lo que hace sobresalir a Telegram es su particular método de encriptación y protocolo de seguridad, ya que no sigue el típico esquema de encriptación PGP para el intercambio de llaves, sino que implementa su propio protocolo llamado MTProto, el cual utiliza algoritmos de poca popularidad, como AES-IGE, y aun así logra asegurar confidencialidad, integridad y autenticidad.

II. Encriptación de Mensajes en Telegram

A. Protocolo encriptación

AES-IGE(Advanced encryption standard - Infinite Garble Extension) es el protocolo de encriptación aplicado el mensaje en MTProto, se basa en CBC(Cifrado bloques en cadena). Pero Telegram utiliza una particular fórmula para encriptar el mensaje, $c_i = f_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$ de tal forma que además de utilizar el mensaje cifrado c_{i-1} como en CBC tradicional, agrega la utilización de m_{i-1} luego de utilizar la llave K en la función f_K(La función f_K es la aplicada de forma genérica por AES). La llave variable para encriptar el mensaje y la llave para el valor inicial de cada bloque de datos vienen dados por KDF.

B. SHA-1

Es una función criptográfica que su entrada la convierte en 160 bit(equivalente a 40 caracteres en hexadecimal) de mensaje digerido, de tal manera de dejar el mensaje en un formato hexadecimal imposible de revertir (más que por fuerza bruta) para brindar integridad al receptor y transmisor.

C. SHA-256

Es una función criptográfica que su entrada la convierte en 256 bit(equivalente a 64 caracteres en hexadecimal) de mensaje digerido, de tal manera de dejar el mensaje en un formato hexadecimal imposible de revertir (más que por fuerza bruta) para brindar integridad al receptor y transmisor.

D. KDF

Una *KDF* o *Key Derivation Function* es una función que se utiliza para derivar una o más claves secretas a partir de otro valor secreto. Las claves resultantes típicamente son más fuertes criptográficamente hablando. En el caso de MTProto 2.0 la KDF está basada en SHA-256 y se utiliza para generar una clave AES y un vector de inicialización AES IGE, ambos de 256 bits, a partir de un fragmento de una clave de autenticación compartida y del MAC obtenido del mensaje.

El procedimiento implementado en MTProto 2.0 es el siguiente:

```
msg_key_large = SHA256(auth_key[88+x:88+x+32]
                       + msg + padding);
msg_key = msg_key_large[8:24];
sha256_a = SHA256(msg_key + auth_key[x:x+36]);
sha256_b = SHA256(auth_key[40+x:40+x+36]
                  + msg_key);
aes_key = sha256_a[0:8] + sha256_b[8:24]
          + sha256_a[24:32];
aes_iv = sha256_b[0:8] + sha256_a[8:24]
          + sha256_b[24:32];
```

En el pseudo-código anterior el operador + se utiliza para concatenar. El valor `x` es 0 para mensajes enviados desde el cliente al servidor y 8 para mensajes enviados desde el servidor al cliente. Los valores de indexación representan bytes.

E. Diffie-Hellman

El intercambio de Diffie-Hellman es un método para establecer un secreto compartido entre dos entidades de forma segura a través de un canal público. De forma resumida el proceso es el siguiente:

- Alicia y Bob se ponen de acuerdo para elegir dos números primos p y g de forma pública.
- Alicia escoge un número secreto a que sólo conocerá ella, calcula $A = g^a \bmod p$ y le envía A a Bob.

- Bob calcula lo mismo, pero con un número secreto b y un resultado $B = g^b \bmod p$. Bob envía B a Alicia.
- Alicia toma el número B que recibió de Bob y realiza el mismo cálculo de nuevo con ese número, es decir $S = B^a \bmod p$
- Bob hace lo mismo con el número que recibió de Alicia: $S = A^b \bmod p$

Es simple demostrar que los números S a los que llegan Alicia y Bob son los mismos ya que:

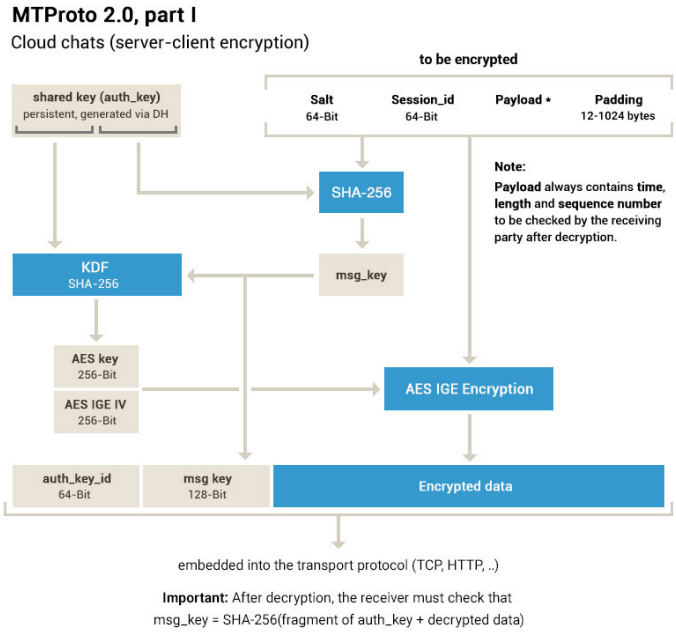
$$(g^a \bmod p)^b \bmod p = g^{ab} \bmod p \quad (1)$$

$$(g^b \bmod p)^a \bmod p = g^{ba} \bmod p \quad (2)$$

Finalmente, habiendo hecho esto Bob y Alicia ahora tienen un secreto compartido, y incluso aunque alguien haya estado leyendo el tráfico, esa persona no puede obtener el secreto generado.

Claro que para que esto funcione bien los valores de p, a y b deben ser apropiados. p en este caso es un valor de 256 bytes, $1 \leqslant a < p$ y $1 \leqslant b < p$. g típicamente es un número entero pequeño como 2, 3, 5, etc.

III. Implementación de Encriptación en Protocolo MTProto



Este es el esquema del protocolo MTProto actual (actualizado a finales de noviembre 2017).

Lo que se envía a través de la red consiste en los datos encriptados junto con un encabezado compuesto por el valor de 64 bits `auth_key_id` (que identifica a la clave de autenticación, compartida por el servidor y el usuario) y el valor de 128 bits `msg_key`.

La clave de autenticación `auth_key` es un valor de 2048 bits obtenido a través de un intercambio Diffie-Hellman entre el servidor y la aplicación cuando esta última se ejecuta por primera vez; es un secreto compartido, nunca se transmite y casi nunca cambia.

La `msg_key` es un valor de 128 bits que puede ser interpretado como un MAC del mensaje. Es obtenido a través de la función hash SHA-256 tomando como entrada un fragmento de 32 bits de la clave de autenticación y el mensaje a encriptar (incluyendo el encabezado y padding), y seleccionando los 128 bits del medio del resultado.

La `msg_key` se combina con otro fragmento de la `auth_key` en la KDF para generar una clave `aes_key` de 256 bits y un vector de inicialización `aes_iv` también de 256 bits. Este procedimiento es descrito en la sección II-D. Estos dos valores se utilizarán para encriptar el mensaje (el cual incluye otros datos además del texto mismo, como sal, id de sesión, padding, entre otros) utilizando encriptación AES-256 en modo *infinite garble extension* (IGE).

Cada mensaje además lleva los siguientes datos que son revisados en la desencriptación para hacer el sistema más robusto ante ataques:

- Sal (64 bits)
- Id de sesión (64 bits)
- Número de secuencia (32 bits)
- Largo del mensaje (32 bits)
- Identificador del mensaje (64 bits)
- Padding (12 a 1024 bytes)

La sal es un número aleatorio de 64 bits que se va cambiando periódicamente cada varias horas a pedido del servidor; su principal función es proteger contra ataques de reproducción y cambios de clock a futuro distante. La id de sesión es un número aleatorio también de 64 bits para distinguir entre sesiones individuales.

El identificador de mensaje es único para el mensaje dentro de la sesión, y también sirve como marca de tiempo: aumentan monotónicamente y dependen del tiempo unix de creación del mensaje: $msg_{id} \approx unixtime \cdot 2^{32}$. Los mensajes si fueron creados más de 300 segundos antes de ser recibidos o más de 30 segundos después de ser recibidos. Además los identificadores de los últimos N mensajes son almacenados; si se recibe un mensaje con un identificador menor a alguno de éstos, es descartado.

El padding es aleatorio, pero debe ser tal que el largo del mensaje resultante sea múltiplo de 16 bytes.

Para desencriptar el mensaje, el receptor primero deriva las claves `aes_key` y `aes_iv` de la misma forma que lo hizo el emisor: con la KDF de la sección II-D y utilizando la `msg_key` (que recibió del emisor) y el fragmento de `auth_key` (que conoce). Luego utiliza estas claves para desencriptar el mensaje.

Una vez obtenido el mensaje desencriptado, se calcula una nueva `msg_key` a partir del mensaje desencriptado y un fragmento de `auth_key` (de la misma forma que lo hizo el emisor) y se compara con la `msg_key` recibida. Si son distintas, el mensaje es descartado. Si no, se procede con otros

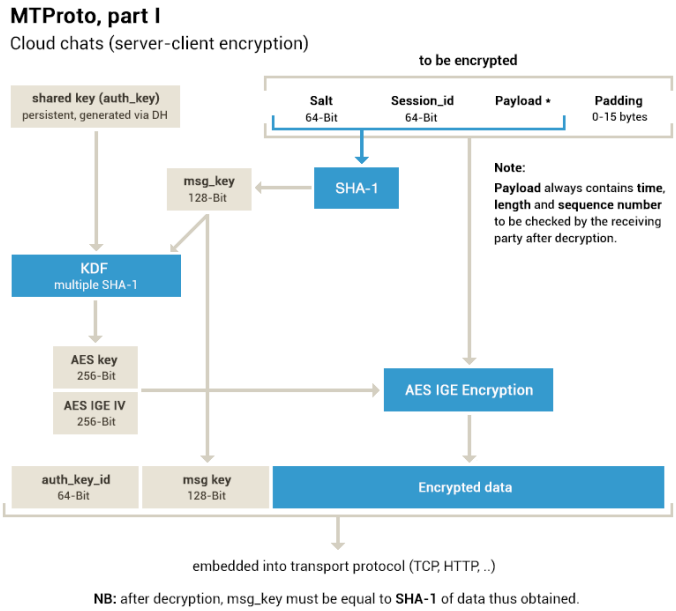
chequeos: Se revisa que el tiempo de creación del mensaje no sea más de 300 segundos menor o 30 segundos mayor que el actual, y que el identificador del mensaje no sea menor a ninguno de los N identificadores de mensajes anteriores almacenados.

Si se desencripta correctamente y se pasan todos los chequeos, se puede concluir que el mensaje es íntegro y auténtico. El mensaje se obtiene fácilmente de la data desencriptada: hay que quitar la sal, id de sesión, padding y todos los otros datos asociados, lo cual es simple ya que se conoce el largo en bits de cada parte.

Lo primero que debe hacer una aplicación cliente es crear una clave de autenticación que normalmente se genera cuando se ejecuta por primera vez y casi nunca cambia. El inconveniente principal del protocolo es que un intruso que intercepta mensajes pasivamente y luego de alguna manera se apropia de la clave de autenticación (por ejemplo, al robar un dispositivo) podrá descifrar todos los mensajes interceptados. Probablemente esto no sea un gran problema (al robar un dispositivo, también se podría obtener acceso a toda la información almacenada en la memoria caché del dispositivo sin descifrar nada); sin embargo, se toman los siguientes pasos para superar esta debilidad:

Para crear la clave de sesión, lo primero que un cliente debe hacer después de crear una nueva sesión es enviar una consulta RPC especial al servidor ("generar clave de sesión") a la que responderá el servidor, después de lo cual todos los mensajes subsiguientes de la sesión se cifran utilizando la clave de sesión también.

Proteger la clave almacenada en el dispositivo cliente con una contraseña (de texto); esta contraseña nunca se almacena en la memoria y la ingresa un usuario al iniciar la aplicación o más frecuentemente (según la configuración de la aplicación). Los datos almacenados (en caché) en el dispositivo del usuario también pueden protegerse mediante cifrado utilizando una clave de autorización que, a su vez, debe protegerse con contraseña. Entonces, se requerirá una contraseña para obtener acceso incluso a esos datos.



Este es el esquema del protocolo MTProto de la versión anterior, que presenta algunas diferencias como el uso de SHA-1 y la no inclusión de la clave de autenticación y el padding en el cálculo de `msg_key`.

IV. Ejemplo de Implementación

Se implementó la encriptación y desencriptación de MTProto 2.0 en el lenguaje Python 3. El código básicamente sigue los pasos descritos en la sección III, pero de forma bastante simplificada. Un programa encriptador crea un mensaje y lo encripta, y un programa desencriptador toma el mensaje encriptado y lo desencripta, además verificando el `msg_key`.

Se utilizó la librería *hashlib* de python para implementar el algoritmo hash SHA-256. Para la encriptación IGE se utilizó la implementación encontrada en [2].

El código de la implementación básicamente sigue los siguientes pasos:

- Creación de clave de autenticación `auth_key` (2048 bits) y el identificador de clave de autencicación `auth_key_id` de forma aleatoria. Es guardada en un archivo de texto al cual deben tener acceso tanto el programa encriptador como el desencriptador.
- El programa encriptador crea un mensaje junto con la sal (aleatoria) e id de sesión (aleatoria) y calcula el largo y padding (aleatorio, pero el largo resultante debe ser múltiplo de 16 bytes). Se concatenan de la siguiente forma: Sal, id de sesión, largo de mensaje, mensaje, padding.
- Se genera la *msg_key* (128bits) con la función `sha256()` de la librería `hashlib`, tomando como entradas la concatenación de un fragmento de 32 bytes de la clave de autorización (partiendo desde el byte 88) y el mensaje incluyendo los datos asociados (sal, id de sesión, largo, padding)

- Se derivan las claves necesarias para la encriptación AES IGE: *aes_key* y *aes_iv* (256 bits c/u) usando la KDF detallada en la sección II-D a partir de `msg_key` y un fragmento de `auth_key`.
- Se encriptan los datos utilizando la función `crypt.ige_encrypt()` obtenida de [2].
- Los datos encriptados junto con `msg_key` y la `auth_key_id` se guardan en un archivo de texto, que luego se debe hacer llegar al programa desencriptador de alguna forma.
- El programa desencriptador lee el archivo de texto creado por el programa encriptador.
- El programa desencriptador deriva las claves AES IGE de la misma forma que el programa encriptador, ya que conoce `auth_key` y recibió `msg_key` junto con los datos encriptados en el archivo de texto.
- Se desencriptan los datos con la función `crypt.ige_decrypt()` encontrada en [2].
- Se obtiene otra `msg_key` a partir de los datos desencriptados y el fragmento de `auth_key` (nuevamente de la misma forma que lo hizo el programa encriptador)
- Se comparan la `msg_key` recibida en el archivo de texto con la que se acaba de calcular. Si son iguales, se asume que el mensaje no fue alterado.
- Se muestra el mensaje desencriptado.

V. Conclusiones

- La implementación de Advanced Encryption Standard (AES), permite generar confidencialidad. El uso de la función hash en SHA-256 genera el paso necesario para la autenticación del mensaje y por ultimo para el receptor al poseer el mensaje encriptado y al mismo tiempo el mensaje digerido, puede corroborar la integridad de este.
- La encriptación AES IGE proporciona una seguridad elevada ante ataques por fuerza bruta al poseer en su algoritmo de cifrado de bloques en cadena, agregando además un or exclusivo del mensaje en una iteración anterior. Por lo tanto para un atacante el ataque basado en texto legible conocido y basado solo en texto cifrado no será suficiente para decodificar el mensaje.

VI. Referencias

References

[1] Documentación Telegram: https://core.telegram.org/mtproto

[2] Sammy Pfeiffer, Anton Grigoryev (2015). Implementación Python AES IGE: https://github.com/Surye/telepy/blob/master/crypt.py