Encryption Report

NG911

Team Members:
Rabaa Abdelrahman,
Li Pan,
Renz Rivero

April 9, 2022

Possible security issues

While securing data communications on all levels is encouraged in almost all software systems, the need to secure the data and protect its confidentiality increases as the data becomes more valuable.

The worst thing that might happen is a server attack, such as a DDoS attack which would paralyze our servers blocking users from using the system, or actual penetration which could leak sensitive information of all users. Our application is stored on Google's firebase database which is relatively secure, as its communications are done on HTTPS. Google's firebase servers are all over the world, although we don't have the ability to change its location after it has been declared, we still can choose any server we want during the initial stage once the databases are created. Servers are one piece of the chain which is relatively hard to go around or break into, given other factors which are easier to penetrate, such as the user's device. Google's ability to handle tons of requests a second gives it a great advantage, attempting a DDoS attack on firebase servers is possible, but extremely expensive, which would not completely eliminate the chances of being DDos'ed but rather make them challenging enough.

The main ways to get to the user's data are as follows:

a. Breaking into the server through our firebase console.
b. Breaking firebase's authentication session management.
c. Man in the middle attack, done by:
    i. Owner of the internet subscription.
    ii. Packets rerouted to an adjacent internet user.
        1. Through an insecure internet connection.
        2. Through a public internet connection.
        3. Through a user on the same internet connection with an access to the router.
    iii. Internet Service Provider/s.

For point 'b', we have to understand how firebase authentication works. On the user's side the authentication is done via phone number and one-time-password "OTP" verification code. Generally the authentication is secured through the SHA-256 key pair stored on the apk release, and the firebase console. While the phone authentication is relatively secure, as it gives access into the system only when the user has his/her own cellular phone, it might not always be the case. It might be fairly easy to go around it if the attacker had access to the user's physical device. In summary, attacking the connection is almost impossible given the SHA-256 keypair. Yet the human factor might play a role in case the phone got stolen or was accessible by the attacker. While we have thought about solutions to this problem, we have found another way which includes a 2 step verification method. Alongside the phone authentication we might have a permanent password for verification entered each time the user opens the application. Given the fact that our system has been created to increase the efficiency of a 911 call, a secure password typed during an emergency situation is not a good idea. "People in an emergency situation might have shivering hands, or might not be able to move their hands at all." stated our emergency advisor officer John Leitch. It is the user's duty to prevent physical unauthorized access to his/her phone.

The defect 'c' is reduced by the secured connection "HTTPS" and SHA-256 keypair. Yet the point 'a' is a potential threat as it is not encrypted anymore when it is stored in the firebase console. Being the initial firebase console administrators, we have unethical authority over confidential information. Thus adding another layer of data encryption to block all unauthorized parties reading the data.

Possible approaches

There are mainly two possible encryption algorithms that have been widely used on flutter.

1. RSA
2. AES

Unlike the AES, RSA is an asymmetric encryption, meaning that it has two different keypairs with algorithms easing one way computation, and making it hard to reverse.

Possible libraries

For RSA we used: rsa_encrypt, encrypt and pointy castle, as they had the highest points and reviews on the pub.dev flutter package website. Rsa_encrypt is an interface to the point castle. RSA key pairs are generated, converted from keypair types to strings and vice versa.

The AES is created using the cryptography library for the same reason mentioned the the previous graph, as well as the examples provided by the developers to use and debug the functions.

Constraints

The encryption must be done without drastically affecting the performance, or eliminating the core functionalities. The transition between the stages as in the storyboard provided on our Github must not be laggy or held back by the process of key-generation.

Proposed solutions

Our first solution was to use RSA only to send and receive data between the clients. The Audio-video is already encrypted on firebase through the package, so we will encrypt the messages.

Testing functionality

We have implemented a test flutter application imitating the communication with encryption between the clients. The user and the dispatcher were two different classes without any shared variable. The dispatcher class must be able to do the following:

1. Generate an RSA key pair.
2. Convert the Public key pair to string and send it to the user class.

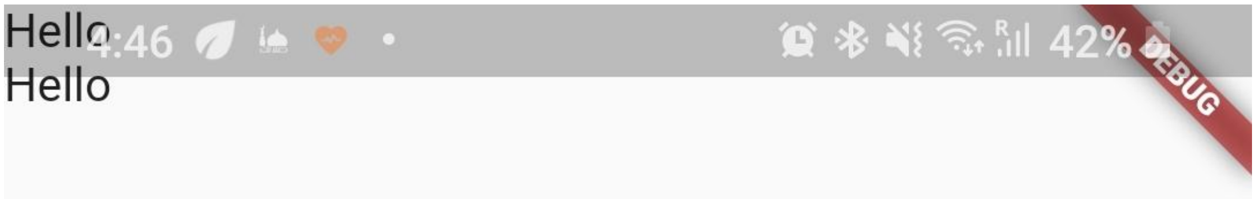
The user class must then

3. Convert the public key string to public key type.
4. Store and display the word “Hello”.
5. Digest the string of the word “Hello” into bytes.
6. Send the encrypted text to the dispatcher class as bytes.

The dispatcher class then will proceed:

7. Use the private key to decrypt the bytes.

8. Convert the bytes into a string.
9. Display “Hello” on the screen.


Hello
Hello

As shown in the figure “” the RSA encryption works perfectly.

Implementation

The implementation has been done using the same way in the application, with the difference that the keypairs and the encrypted text are communicated through firebase instead of mediator variables in the main function. Once the screens get initialized, the keypairs are exchanged. The sender function which is used to send plain text will encrypt the text instead before sending. The listviewer which is used to display the chat is then calling its own function which decrypts the text before displaying.

Problems faced

The private key of key pair ‘X’ can only decrypt what has been encrypted by the public key of the key pair ‘X’. That means that if Someone sends a message, and sends it to the database encrypted with the other side’s public key, he/she will not be able to decrypt it as they don’t have the private key. That means that they have to save it locally before encrypting it.

The other problem is that the dispatcher side surprisingly took minutes to generate the key, as flutter web is not as fast, we had to store a fixed key pair on the dispatcher’s end instead, which is not a good idea. Lastly, the RSA turns out a good encryption algorithm in case the message encrypted is not long, and of a known length. If the data to be encrypted are not uniform, RSA may crash as it will not handle large amounts of data. To overcome this, we either should have the keys of larger numbers, which will increase our key generation time exponentially, or to change the encryption algorithm.

Implementation - 2

The Advanced Encryption Standard, or the AES, is another secure symmetric encryption algorithm. Unlike RSA, it doesn't depend on irreversible functions. The AES depends on matrix operations rather than prime numbers math, which makes it faster. The encryption process is based on a secret box, ciphertext. The secret box has two attributes, the nonce and the mac. Similar to what we have done with the RSA, we have tried the AES on a test sample and it worked.

The worst problem experienced while encrypting is the technicality of decryption, integrated with the message listing on the UI. Flutter lists the query using the "ListviewBuilder" widget. Which nested in a "StreamBuilder" widget. The StreamBuilder streams new data coming from the query and passes it to the ListviewBuilder to display. For flutter to operate normally, the query and the stream must be initialized before the loading of the page, then it will keep updating once a new stream element is passed. Initializing the stream and the query after the page build will cause the page to keep firing, "refreshing" each part of the second.

To decrypt a message, the ciphertext must be sent to a function which operates asynchronously. An asynchronous function is a function that promises to return a value in the future. The ciphertext is then returned into the displayed message. The asynchronous function is not to be called within the ListviewBuilder; it must be used in a FutureBuilder. The FutureBuilder faces the same issue of refreshing the page, but in this case, having the messages initialized before the page build. To fix that, we implemented each individual message to be a different stateful widget to be created separately, to be instantiated and disposed as needed. As the messages may experience milliseconds - 1 second of lag, it is not bad given the fact that messages are now encrypted successfully.

We added RSA to the encrypt the AES, which made the flutter web slow down clearly. We decided to stop encrypting more stuff and to depend solely on AES to increase the efficiency. Encrypting tons of data with a live stream may be not bad on a php-driven server, but flutter web adds latency and lowers the response time. We now have all communication “text”, “audio”, and “video” encrypted with AES symmetric keys.

Limit testing

We tested sending large text messages with special characters on the messages. It works perfectly without crashing like the RSA.

Hashing

Hashing is a small portion of our system, as per the firebase authentication limitations, regular usernames and passwords are not supported. We implemented the logic of the authentication using usernames and passwords stored in the firebase database. The passwords then are hashed using the SHA-256, which is then repeated whenever there is a login and compared to the hashed value stored in the database for verification.

Future implementation

For future implementations, we would host a server on AWS with a dedicated API and a secure connection to deal with heavy data operations, key pair generation, and data management. The BLOC, or the business logic would be implemented to make the process of data management and streaming more organized, thus more efficient. Encryption could then be done on all data. The changes needed to enhance the encryption would add one more month to fully operate, with 15 more days of debugging and continuous testing.