Vaibhav, 2024, 12:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology

An Open Access Journal

Developing an End-to-End Secure Chat Application

Vaibhav, Bhaskar Chauhan
Department of Computer Science & Engineering,
Dronacharya Group of Institutions, Greater Noida, India

Abstract- Chat applications have emerged as indispensable tools on smartphones, offering users the ability to exchange text messages, images, and files at no cost. However, ensuring the security of these communications is paramount. This paper proposes a secure chat application that implements End-to-End encryption, safeguarding private information exchanged between users and providing robust data protection. The application also addresses storage security concerns. By outlining a set of requirements for a secure chat application, this paper informs the design process. The proposed application is evaluated against these requirements and compared with existing popular alternatives to assess its security features. Furthermore, the application undergoes rigorous testing to validate its End-to-End security capabilities

Keywords- Secure chat application, Security End to end encryption, Data protection

I. INTRODUCTION

With the rapid proliferation of mobile devices, smartphones have seamlessly integrated into our daily lives. Chat applications, in particular, have undergone significant evolution, reshaping the landscape of social media with their compelling features [1]. These apps provide real-time messaging capabilities and a plethora of services, including text exchanges, media sharing, and file transfers. Furthermore, they offer cross-platform compatibility, catering to users across Web, Android and IOS ecosystems. Currently, these chat apps boast hundreds of millions of monthly active users [2].

These applications typically adopt one of two architectures: client-server or peer-to-peer networks. In a peer-to-peer network, there's no central server, and each user manages their data storage independently. Conversely, client-server networks rely on dedicated servers to facilitate communication, with data centralized on these servers [3].

However, despite their widespread adoption, security and privacy in chat applications remain critical yet often overlooked aspects. A study conducted by the Electronic Frontier Foundation revealed that many popular messaging apps fall short in meeting basic security standards. This laxity raises concerns about potential misuse of user conversations for various purposes. The prospect of private conversations being accessible to unauthorized entities is particularly alarming from a privacy standpoint.

While many applications rely solely on Transport Layer Security (TLS) to secure communication channels, this approach leaves messages vulnerable to interception by service providers or attackers [4]. To uphold privacy and protection, messages should be encrypted end-to-end, ensuring that only the sender and receiver can access the content, thereby safeguarding against third-party intrusion. Additionally, ensuring the security of local storage on devices adds another layer of protection [5].

In our research paper, we aim to address these concerns by proposing an end-to-end security framework. This framework not only guarantees

that only authorized parties can access messages but also prioritizes storage protection and facilitates fast message transfers between users. By emphasizing security, privacy, and speed, our proposed framework seeks to enhance the overall user experience while mitigating risks associated with communication in chat applications.

The main contributions of this paper are the following:

- Propose client-server mobile chat application which supports the status of the communicating parties whether online or offline.
- Provide a friendship request service.
- Secure key exchange, then calculate the session key.
- Secure exchange of end-to-end messages.
- Analysis and Test the proposed chat.

II. CHAT APPLICATIONS

In this section, we provide a concise overview of several widely-used chat applications available in the market, focusing on their respective security and privacy considerations. Regrettably, the lack of public availability or open-source nature of certain chat applications poses challenges for evaluation by developer communities, security experts, or academic researchers.

1. Viber

Viber is a popular instant messaging and Voice over IP (VoIP) application designed for smartphones, developed by Viber Media. It allows users to exchange messages, images, videos, and audio media messages. Recently, Viber introduced end-to-end encryption for one-to-one and group conversations, provided that all participants are using the latest Viber version 6.0 for Android, iOS, or Windows 10. However, there are some limitations to this encryption, particularly regarding attachments sent via the iOS Share Extension on iPhone and iPad, which are not currently supported [6].

Despite its encryption efforts, Viber has faced criticism regarding privacy issues, such as adding contacts without their explicit consent or adding them to groups without permission. Additionally, concerns have been raised about the security of locally stored data. One significant challenge is that Viber is not open source, making it difficult for independent evaluation and scrutiny of its security practices.

2. WhatsApp

WhatsApp is one of the most popular messaging application, recently enabled end-to-end encryption for its 1 billion users across all platforms. WhatsApp uses part of a security protocol developed by Open Whisper System, so provides a security-verification code that can share with a contact to ensure that the conversation is encrypted [7]. It is difficult to trust in WhatsApp application completely because the application is not open source, making it difficult to verify the functioning process and match them with the work of the encryption protocol which was announced.

3. Telegram

Telegram is an open source instant messaging service enables users to send messages, photos, videos, stickers and files [8]. Telegram provides two modes of messaging is regular chat and secret chat. Regular chat is client- server based on cloud-based messaging, it does not provide end-to-end encryption, stores all messages on its servers and synchronizes with all user devices [9]. More, local storage is not encrypted by default. Secret chat is client-client provides end-to-end encryption. Contrary to regular chat messages, messages that are sent in a secret chat can only be accessed on the device that has been initiated a secret chat and the device that has been accepted a secret chat they cannot be accessed on other devices. Messages sent within secret chats can be deleted at any time and can optionally self-destruct [8].

Telegram uses its own cryptographic protocol MTProto, and has been criticized by a significant part of the cryptographic community about its security[9].

The registration process of Telegram, Viber and WhatsApp depend on SMS. SMS is transported via

Signaling System 7 (SS7) protocol. The vulnerability lies in SS7 [10]. Attackers exploited SS7 protocol to login into victim's account by intercepting SMS messages [11]. Because of Telegram cloud-based, the attacker exploits it and makes full control of the victim account and can prevent him to enter into his account. To make the account more secure should activate two-factor authentication [12].

4. Facebook Messenger

Facebook Messenger is a popular messaging service available for Android and iOS. It provides two modes of messaging is regular chat and secret conversations. Regular chat does not provide end-to-end encryption only secure communication by using TLS, and it stores all messages on its servers. Secret conversations have the same idea of Telegram secret chat [13].

III. PROPOSED ARCHITECTURE

1. Secure Chat Requirements

In this section, we propose a set of requirements to make secure chat application:

- Password stored on the chat server should be encrypted.
- Providing either secure session or TLS. Secure session is a unique key for each session. Ensures that communication is with the right person and no man-in- the-middle can read the messages.
- Messages must be encrypted to maintain security and privacy.
- Local storage must be protected by encryption.
- Messages are not stored on the chat server but stored on the user's device.
- It is not allowed to exchange messages if they are not friends.

2. Proposed Architecture

The proposed architecture is designed to be Client-Server chat application. In client side, when a user sets up the application, the user either selects registration or log-in. In server side, the chat server consists of users' server and a message server. User's server that manages user's credentials. Message server handles messages between users by using Firebase Cloud Messaging (FCM).If the recipient is offline, the messages will be stored temporarily on the FCM queue for a specific period of time, and when recipient becomes online these messages are forwarded to him then deleted from the queue. The generic architecture is shown in Fig. 1

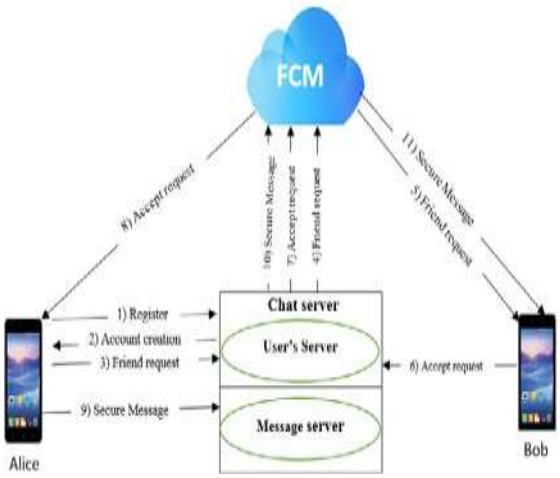

Fig 1: Generic Architecture of Proposed Chat.

3. Registration an Account

Before starting the application, there must have a lock screen to configure the Keystore that provides a secure container to store the local storage key to make more difficult for extraction it from the device by unauthorized persons or other applications [14]. Each account has only one device and it is distinguished by device id. In addition, Email and username are unique. Name, email and password are required to register a new account. After typing the registration information, the password is encrypted by using XSalsa20 algorithm [15]then the user credentials are sent to the server. After verification, the server generates a unique identifier that acts as the user ID. After that, the acknowledgement message is received for successful registration to the client application and the client information is stored in local storage.

The application generates a set of keys:

- Key for encrypting the password.
- A public key pair for calculating session key.
- Symmetric storage key for encrypting/decrypting local storage contains contact list, chat history and key store.

4. Login

Email and password are required for user authentication. After typing the authentication information, the password is encrypted then the user credentials are sent to the server. The server checks if the email and password are valid. After validation, JSON Web Token (JWT) [16] is created and sends to the client to store it. When a client makes a request at the later time, JWT is passed with the request. The server verifies of the JWT, if it is valid, the request is processed (Fig.2).

Fig 2: Login process

5. Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) is a service that facilitates messaging between mobile applications and server applications. It's built on Google Play Services that supports cross-platform (iOS, Android & Web). It is a free service that allows sending lightweight messages from the server to the devices whenever there is new data available[17]. This saves a lot of user's battery by avoiding requesting to the server for new messages. It provides TLS for securing channel.

At the beginning of running the application for the first time gets the following:

- The application connects to FCM server and registers itself.
- When successful registration, FCM provides registration token to the device. This registration token uniquely identifies each device.
- The application sends the registration token to the server to store it in MongoDB database.

The above steps are shown in Fig. 3.

Fig 3: Firebase Cloud Messaging

When the server sends a push notification, it sends a request to FCM sending the push message along with the registration token. FCM identifies the target device by using registration token then starts to push data.

6. Session key Setup

To add users to contact list either by username or by email address.

For sending a request to a friend on the assumption that the first user knows the username or email of the second user due to the username and email are a unique for each user and the second user should have already registered in the server. Presumably, the first user is called Alice and the second is called Bob.

When the send request, Bob name is typed by Alice and her public key is fetched from the local storage then the request is sent to the server.

When a request is received, it appears as a notification (Fig. 4). If the friendship request is accepted by Bob, his private key is fetched with Alice's public key to calculate the session key by using Elliptic Curve Diffie-Hellman (ECDH) over the curve Curve25519 [18] and hashes the result with HSalsa20 [15] then the session key is stored in local storage (Fig. 5). In the end, the acceptance is sent with his public key to the server to be delivered to Alice. Upon receipt of the acceptance of the request, the same steps on the above are taken. The session key is calculated by using Alice private key and Bob public key then it is stored in the local storage for later use.

The session key is the same for both parties and this is the strength of the Elliptic Curve Diffie-Hellman (ECDH) and thus it is difficult to attack by the man-in-the-middle. In addition to, the weakness of the traditional Diffie-Hellman has been eliminated.

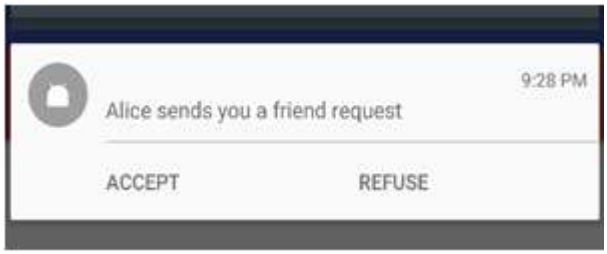

Fig. 4: Friend request notification

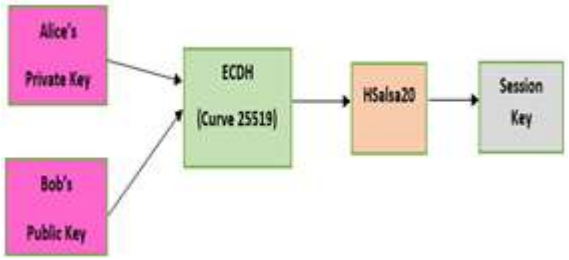

Fig. 5: Session key Setup.

7. Exchanging Messages

When a message is typed, the application encrypts the message using XSalsa20 encryption algorithm to encrypt the message body and Poly1305 to compute a Message Authentication Code (MAC) [19]. Each message has its own separate key and nonce which brings better security for each single message in such discovering one of the keys cannot decrypt previous messages. After encrypting the message, it is encrypted again using the recipient's session key then it is sent to the server (Fig. 6).

After the message is received from FCM, the MAC of the encrypted message is calculated and compares it with the received MAC to verify the integrity of the message. If the results are not the same, it is rejected and does not show to the user otherwise it is decrypted by the sender session key. Next, the message body is verified in the same steps above. Now the key and nonce to decrypt the message are known. The message is then decrypted and stored in the local storage and displayed to the recipient.

If the application is in the background the message will be displayed as a notification while if the recipient uses the application it will be displayed in the chat window.

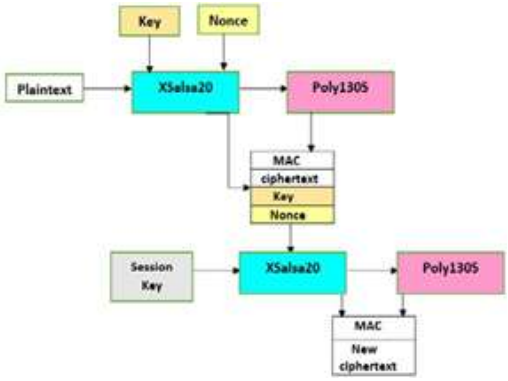

Fig. 6: Procedure to encrypt a message

8. Local Storage

The data is stored locally in the application by using Realm database. Realm is a lightweight mobile database that supports cross-platform. It's easy to use and fast. More, it has lots of modern features such as JavaScript Object Notation (JSON) support, a fluent API, data change notifications and encryption support [20]. Encrypted data is protected from unauthorized access and is accessible only if have been a right encryption key. Realm uses AES-256+SHA2 algorithm and 64-byte key for encrypting storage [21]. To prepare Realm storage passes through several steps that are:

- The application checks whether the lock screen is present or not. If it exists, the following steps are completed.
- Generate Realm Key that is used for encrypting storage.
- Generate key from Keystore.
- Realm key is encrypted with the key generated in step 3 by using AES in CBC mode.
- Save the encrypted key in shared preferences in private mode so that other applications cannot access this data directory.
- Three files are stored in the local storage. User Info file that stores all information pertaining to the user. While Friends file stores all information pertaining to the friends. Finally, Messages file stores all information pertaining to messages.

9. Server Side Implementation

Server-side has relied on Node JS[22] and MongoDB database[23]. Node JS is fast, capable of handling a large number of simultaneous connections with high throughput, which is equivalent to high scalability. MongoDB and Node JS have often used together because of their usingJSON so no need to spend time for transforming the data between them making it easy to deal with each other. In addition, MongoDB provides TLS that makes a secure connection (Fig. 7).

To perform a client request passes through several steps that are:

- Initially, must run the MongoDB connection then run the Node JS from Command Prompt. At this stage, the server is ready to receive the client's request.
- When the client sends a request, the server receives the HTTP request in JSON format. The request then parsed.
- The HTTP request is compared with the base path if it is matched, it is handed to Express framework.
- The Express receives the HTTP request and routes it to the specific endpoint that matched it. In case of not matched with any of the routes will display error in Command Prompt. Otherwise, it will be forwarded to the controller which handles the required function.
- Make a request to MongoDB database by mongoose for processing function.
- When the data is fetched from MongoDB database and the required operations are done, Node JS receives the response then sends to the client.

The above steps are shown in Fig. 8.

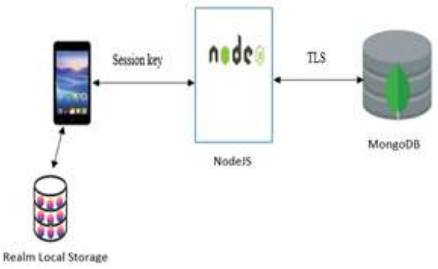

Fig. 7: The Specific Architecture of Proposed Chat.

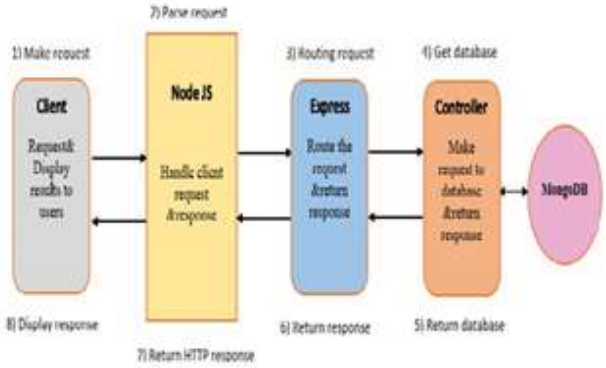

Fig. 8: Implementation of a client request

IV. ANALYSIS THE PROPOSED CHAT

In section 3, we listed a set of requirements for securing chat. To analyze and evaluate proposed chat we have compared proposed chat with popular applications discussed in section 2. The comparison is based on the requirements listed in Table 1.

Table 1: Comparison with Popular Chat Applications

Criteria	WhatsApp	Viber	Telegram	Facebook Messenger	Proposed Chat
Req1	N	N	N	N	Y
Req2	Y	Y	Y	Y	Y
Req3	Y	Y	P	P	Y
Req4	Y	N	N	P	Y
Req5	Y	Y	N	N	Y
Req6	N	N	N	N	Y

Note:"Y" it means that it meets the requirement. "N"does not support the requirement. "P"only the secret part supports it.

V. CONCLUSION

In this manuscript, we outlined a framework aimed at safeguarding the security and privacy of the chat platform. We delineated a series of prerequisites necessary for crafting a secure chat environment, implementing contemporary methodologies, and employing lightweight solutions to ensure swift performance and robust protection for users. The XSalsa20 encryption algorithm emerges as

particularly well-suited for mobile devices due to its robust security features, optimal performance, and minimal impact on battery longevity. Users can rest assured that their messages remain entirely confidential, even in the event of unauthorized access to the mobile device or attempts to breach the application's local data storage.

REFERENCES

1. Ash Read, "How Messaging Apps Are Changing Social Media," 2016. [Online]. Available: https://blog.bufferapp.com/messaging-apps.
2. Most popular messaging apps 2017 | Statista," 2017. [Online]. Available: https://www.statista.com/statistics/258749/most-popular- global-mobile-messenger-apps/.
3. D. Moltchanov, "Client/server and peer-to-peer models: basic concepts," 2013.
4. Martin Kleppmann, "The Investigatory Powers Bill would increase cybercrime — Martin Kleppmann's blog," 2015. [Online]. Available: https://martin.kleppmann.com/2015/11/10/investigatory- powers-bill.html.
5. D. P. Roel Hartman, Christian Rokitta, Oracle Application Express for Mobile Web Applications - Roel Hartman, Christian Rokitta, David Peake - Google Books. 2013.
6. Viber Encryption Overview." [Online]. Available: https://www.viber.com/security-overview/.
7. WhatsApp inc, "WhatsApp security whitepaper," p. 10, 2017.