

COMP3334 Group Project Report: Secure E2EE Chat Web Application

21094823d - Gabriel Conte Chan

21100685d - Zhao Letao

21099695d - LU Zhoudao

19086159d - Shin Hyuk

21100038d YUAN Yunchen

Contents

1. Introduction	2
2. Libraries	2
3. Threat Model	3
3.1 E2E Encrypted chat application	3
3.2 Threats	3
3.3 Countermeasures	4
4. Technical Implementation	5
4.1 Authentication	5
4.1.1. Login	5
4.1.2. Memorized Secrets	6
4.1.3. Rate limiting mechanisms	7
4.1.4. Account Registration and Bind OTP	7
4.1.5. Changing authenticators	9
4.1.6. Session Timeout	10
4.2 E2E Encryption	11
4.2.1. ECDH Exchange	11
4.2.2. Deriving key MAC from the shared secret using	12
4.2.3. Message Format, Encryption and Sending	13
4.2.4. Loading Previous and New Messages	14
4.2.5. Refresh	14
4.2.6. SQL Injections, XSS, and CSFR	16
4.3 TLS	17
4.3.1. Nginx Configuration File	17
4.3.2 Domain Certificate	18
5. Summary	19
References	19

1. Introduction

This report details the development of a secure, end-to-end encrypted (E2E) chat web application that prioritizes user privacy by ensuring messages remain confidential and accessible only to intended recipients. The application incorporates the following security features:

- **End-to-end encryption:** Messages are encrypted AES-GCM. This ensures that only authorized users with the appropriate credentials and keys can decrypt them.
- **Multi-Factor Authentication:** A layered approach to authentication is implemented, combining memorized secrets with One-Time Passwords (OTP). This significantly strengthens account security and minimizes the risk of unauthorized access.
- **Secure Key Exchange:** The Elliptic Curve Diffie-Hellman (ECDH) protocol is employed for secure key exchange. This eliminates the possibility of eavesdroppers intercepting sensitive key information during the communication process.
- **Transport Layer Security (TLS):** Communication between user devices and our servers is encrypted using TLS. This additional layer of security safeguards data in transit, protecting it from potential interception attempts.
- **Website Authentication:** A valid digital certificate is used to authenticate our web application. This critical measure assures users that they are interacting with a legitimate platform and not a malicious imposter site.

2. Libraries

Library	Use
bcrypt	For secure password hashing
Crypto	For key derivation, encryption
base64	For base64 encoding
pyqrcode	For binding the qr code of google authenticator
onetimepass	To validate the totp
re	For regular expressions during input validation

3. Threat Model

Our threat model will identify the security goals of the end-to-end encrypted chat web application, possible attackers, threats, and countermeasures, referencing from the Threat Model of OAuth 2.0 [1].

3.1 E2E Encrypted chat application

As a chat web application, we have the following components:

- Client (the users)
- Server

Security Goals:

- Authentication: The application can verify the user's identity for the session
- E2E Encryption: Chat is encrypted from one client to another to ensure confidentiality, and messages can be decrypted by each user. Ensuring integrity and confidentiality, non-repudiation.
- TLS: Communications should be encrypted in traffic, and the web application should be certified with its own certificate.

3.2 Threats

The table below contains potential threats to our end-to-end encrypted chat web application.

Threat	Description	Impact
Eavesdropping	Attackers intercept messages during the transmission, being able to see their contents or even modify them.	Loss of confidentiality, Integrity, authentication
Cross-Site Scripting (XSS)	Attackers can use XSS to send a malicious script to other users. The end user's browser has no way of knowing that the script should not be trusted, and will execute the script.	Loss of confidentiality, integrity, impersonation

Denial-of-Service (DoS)	Attackers disrupt the chat application by flooding the server with requests	Users are unable to send or receive messages
SQL Injections	Attackers can exploit vulnerabilities to manipulate SQL queries and gain unauthorized access to or manipulate data stored in the database	Data theft, data manipulation, and even server breach
Cross-Site Forgery Requests	Attackers trick an authenticated user's browser into performing unauthorized actions on the website	Unauthorized actions, privacy violation

3.3 Countermeasures

Threat	Countermeasures in our web application
Eavesdropping	E2E encryption with AES-GCM and HMAC to ensure messages are unreadable even if intercepted, and also authenticating their origin and integrity.
Client-Side Attacks (XSS)	Server-side and client-side input validation is done using HTTPOnly flag and Secure on cookies. Cookies will only be transmitted through secure channels and blocked from JavaScript are only included in requests to the domain.
Denial-of-Service (DoS)	Rate limiting for login and registration using Recaptcha, and request IP tracking to limit unsuccessful attempts.
SQL Injections	Server-side and client-side input validation and parameterized SQL queries.
Cross-Site Forgery Requests	Using SameSite cookie to 'Strict', controlling that cookies are not sent with cross-origin requests and originate only from the same site.

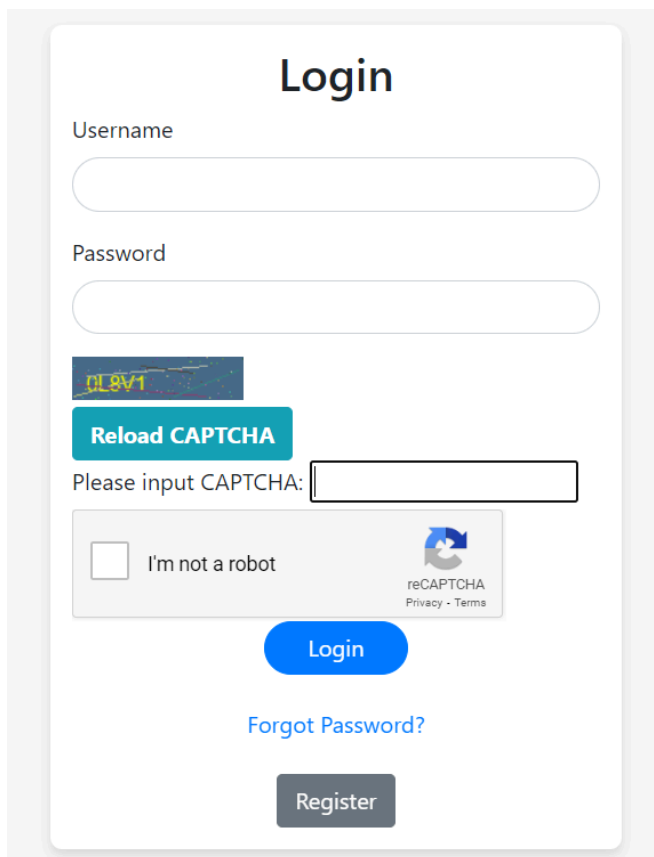
4. Technical Implementation

4.1 Authentication

Secure user authentication ensures that only authorized users can access the platform, perform actions, and access information. Here's a breakdown of our implemented authentication methods

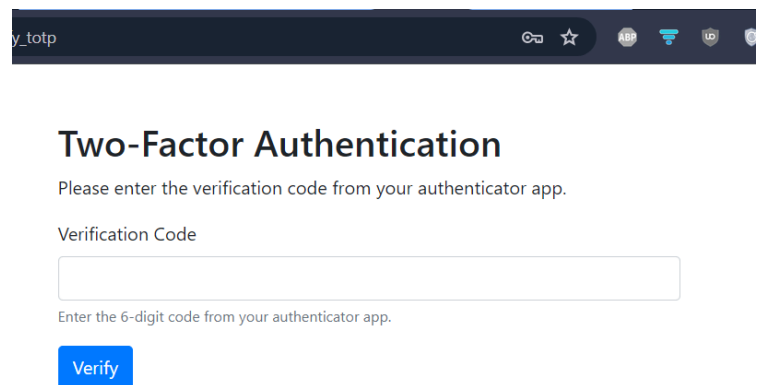
4.1.1. Login

To log in, the user needs to validate their credentials by entering their information. The server will then validate the password is correct by comparing the hashes of the user input and the stored hash relating to the username. Later, the user will be asked to verify using their one-time password.



The login form is titled "Login" and contains the following elements:

- Username:** A text input field.
- Password:** A text input field.
- captcha:** A CAPTCHA image showing the text "0L8V1". Below it is a "Reload CAPTCHA" button.
- reCAPTCHA:** A checkbox labeled "I'm not a robot" next to a reCAPTCHA logo and "Privacy - Terms" link.
- Login:** A blue button.
- Forgot Password?:** A blue link.
- Register:** A grey button.



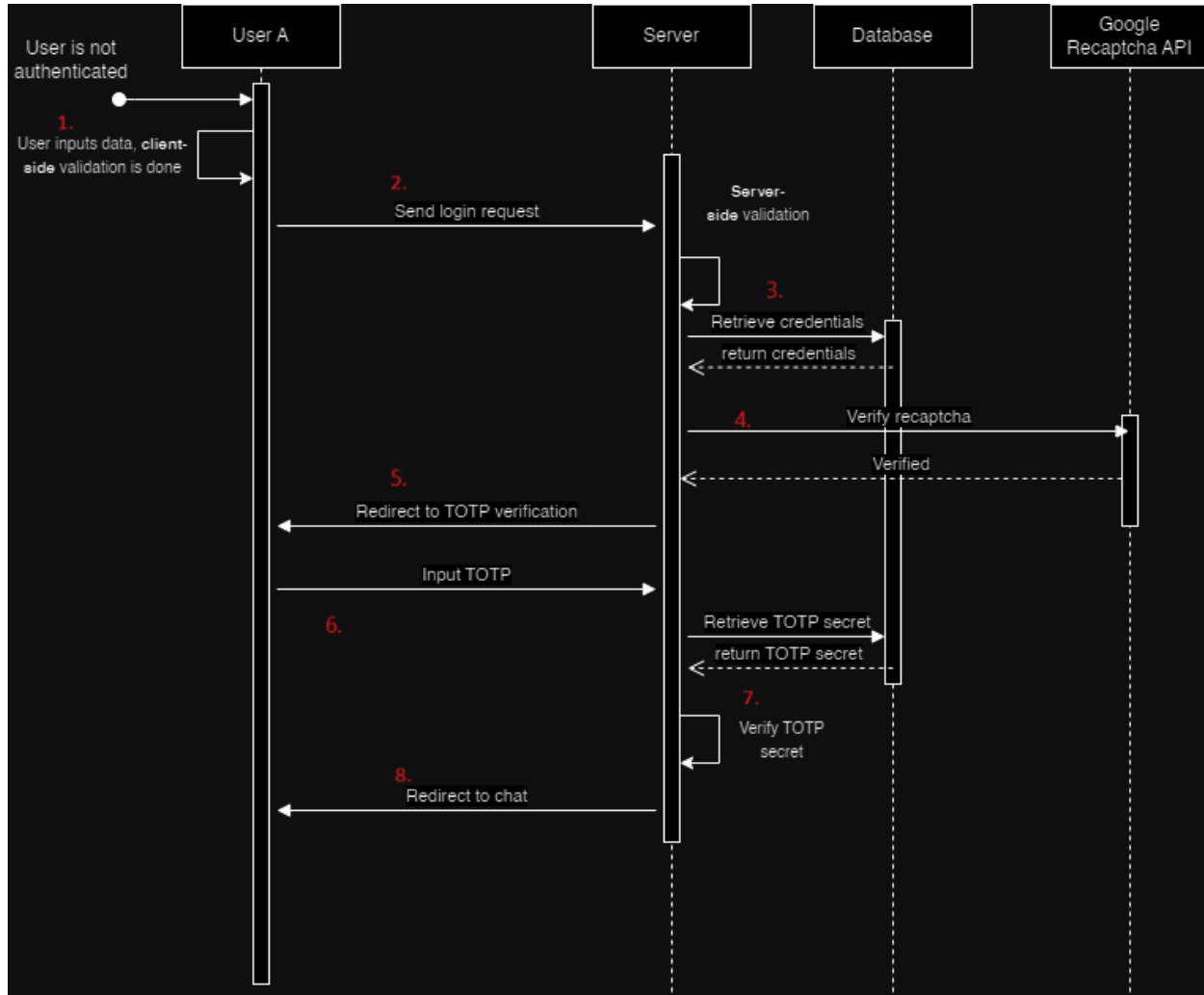
The Two-Factor Authentication form is titled "Two-Factor Authentication" and contains the following elements:

- Instructions:** "Please enter the verification code from your authenticator app."
- Verification Code:** A text input field.
- Instructions:** "Enter the 6-digit code from your authenticator app."
- Verify:** A blue button.

The diagram below displays a successful login attempt from User A.

1. User inputs data. Data is client-side validated so the user sends valid data to the server (e.g., no empty fields).

2. User A sends the request to the server
3. The server validates the data (for security and password validity). Then, User A's credentials are retrieved from the database.
4. The server validates the Recaptcha by asking the Google Recaptcha API.
5. The server redirects the client to the TOTP verification
6. User A inputs the TOTP and sends it to the server
7. The server verifies the TOTP secret with the database
8. The server verifies the user, and the user logs in successfully

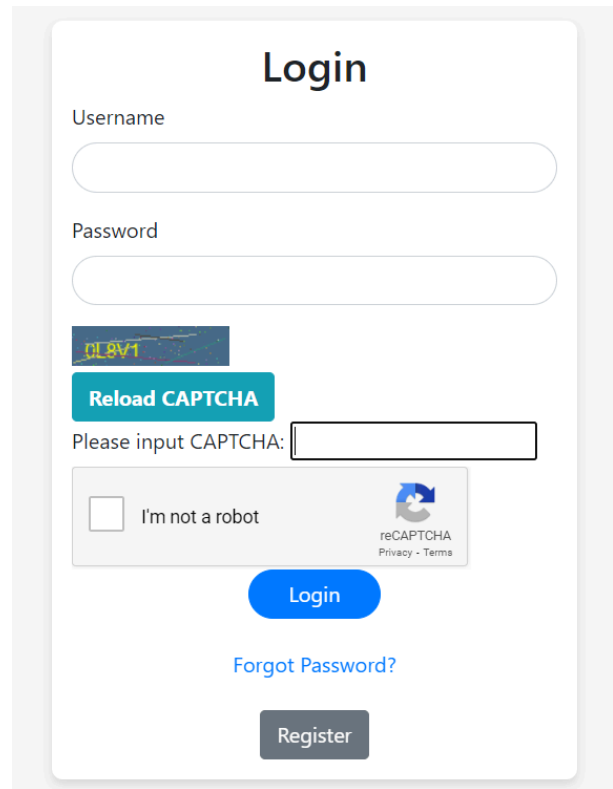


4.1.2. Memorized Secrets

In our application, the user has multiple security secrets such as the password, a security question, and answer, a memorized secret (extra password), and a totp secret. The password is hashed using the bcrypt hash algorithm, and the security question, answer, and memorized secret are encrypted with the user's secret key and IV. The user will also have some public and non-encrypted information, such as the public key and their IV.

4.1.3. Rate limiting mechanisms

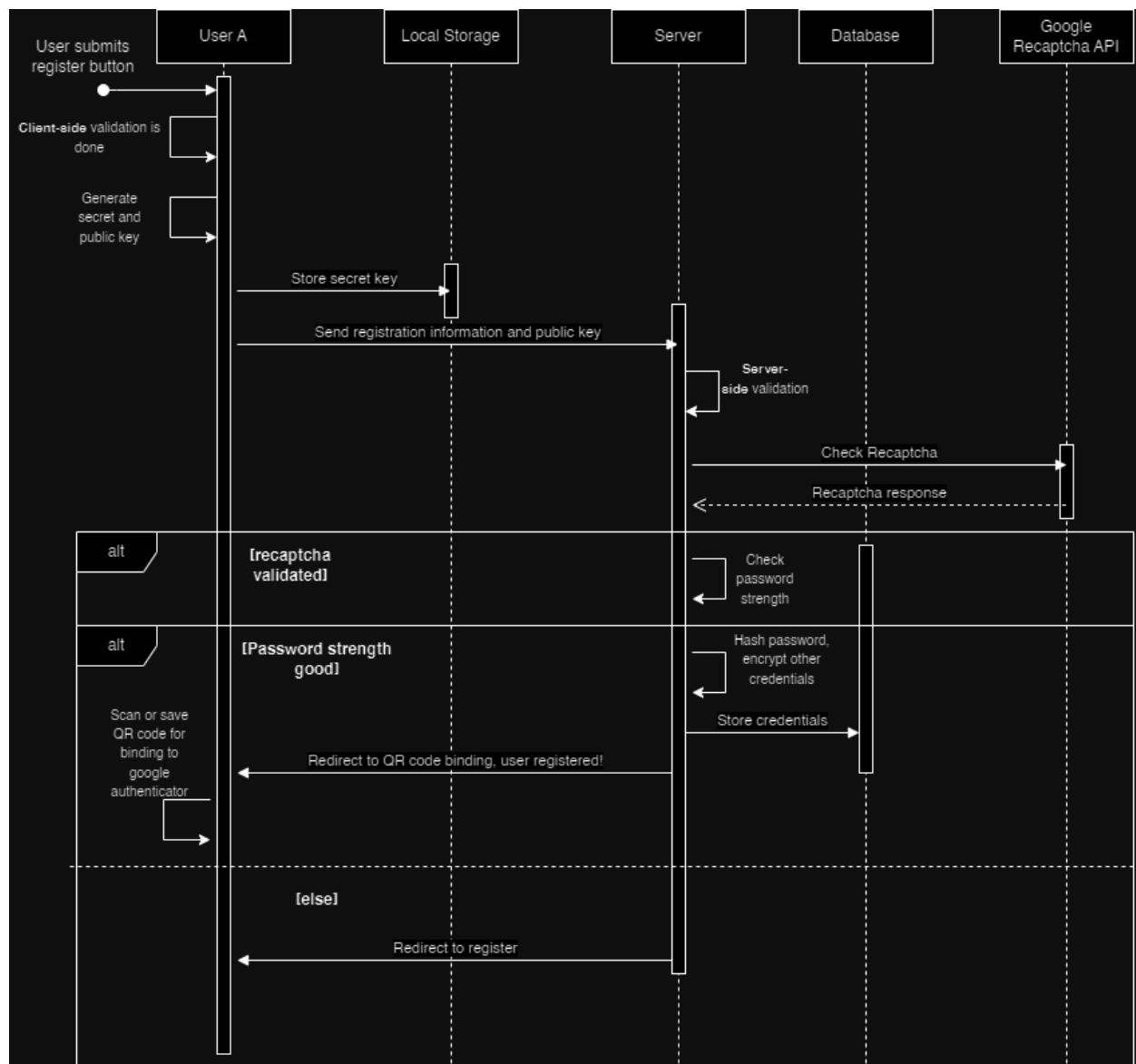
We implement rate-limiting mechanisms to mitigate brute-force attacks that attempt to guess user passwords. These mechanisms limit the speed a user can attempt to log in to an account. For our secure chat web application, we implemented password captchas and image captchas to verify the user is a human. This mechanism also limits the brute force attacks, as it would stop machines from trying values one right after one.



The image shows a login form titled "Login". It contains two input fields for "Username" and "Password". Below the password field is a CAPTCHA image showing the word "ELSVI" in a distorted font. A blue button labeled "Reload CAPTCHA" is positioned below the image. Below the CAPTCHA image is a text input field with the label "Please input CAPTCHA:". Below this field is a reCAPTCHA widget with a checkbox and the text "I'm not a robot". To the right of the checkbox is the reCAPTCHA logo and the text "reCAPTCHA Privacy - Terms". Below the reCAPTCHA widget is a blue button labeled "Login". Below the "Login" button is a link labeled "Forgot Password?". At the bottom of the form is a grey button labeled "Register".

4.1.4. Account Registration and Bind OTP

During registration, users create a new account using a username, password, security question, security answer, public key, iv, totp secret, and memorized secret. Below is a diagram representing how user registration is done in this E2E encrypted chat application.



1. First, the user will input the credentials, and the client will validate the input data
2. The client generates their private and public key
3. Their private key is stored in local storage
4. The client sends the public key and the rest of the data to the server
5. The server will redirect and provide server-side validation
6. If the data is correct and free of attacks, the user will be redirected to the QR binding page.
7. The user scans a QR code that binds the account to a Google authenticator for the totp.
8. The server hashes the user's password using bcrypt
9. After these steps, the server will encrypt the private information, **as explained in chapter 4.1.2.**

10. Finally, the user information will be stored in a MySQL database using the following schema:

users	
PK	<u>user_id</u> INT AUTO INCREMENT
	username VARCHAR(255) NOT NULL UNIQUE
	password VARCHAR(255) NOT NULL, -- Stored as bcrypt hash
	security_question VARCHAR(255) NOT NULL
	security_answer VARCHAR(255) NOT NULL
	public_key VARCHAR(2048), -- ECDH public key
	iv VARBINARY(32), -- For AES CBC
	totp_secret VARCHAR(16) NOT NULL
	memorized_secret VARCHAR(255) NOT NULL
	created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

4.1.5. Changing authenticators

Users can change their memorized secrets for situations they forget their password or for security practices like changing their passwords constantly.

1. The user first inputs their username, with their security question, the answer for the security question, and the memorized secret
2. If correct, they will proceed to enter the new password.
3. Then, they will be redirected to log in, and their new password will be saved.

The screenshot shows a web browser window with the address bar displaying 'group-10.comp3334.xavier28c.fr'. The page title is 'Forgot Password'. The form content is as follows:

Forgot Password
Please provide your username and answer to your security question.

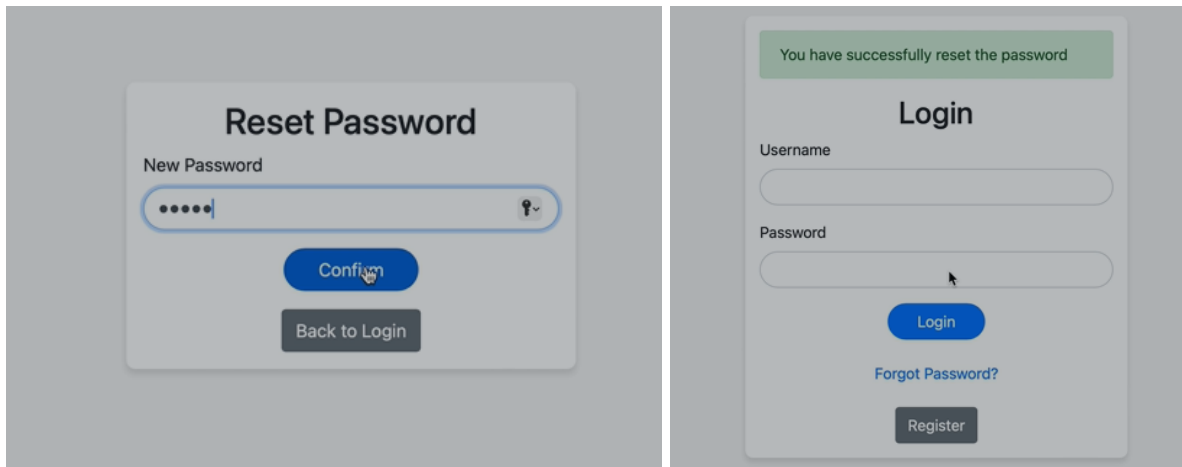
Username:

Security Question:

Answer:

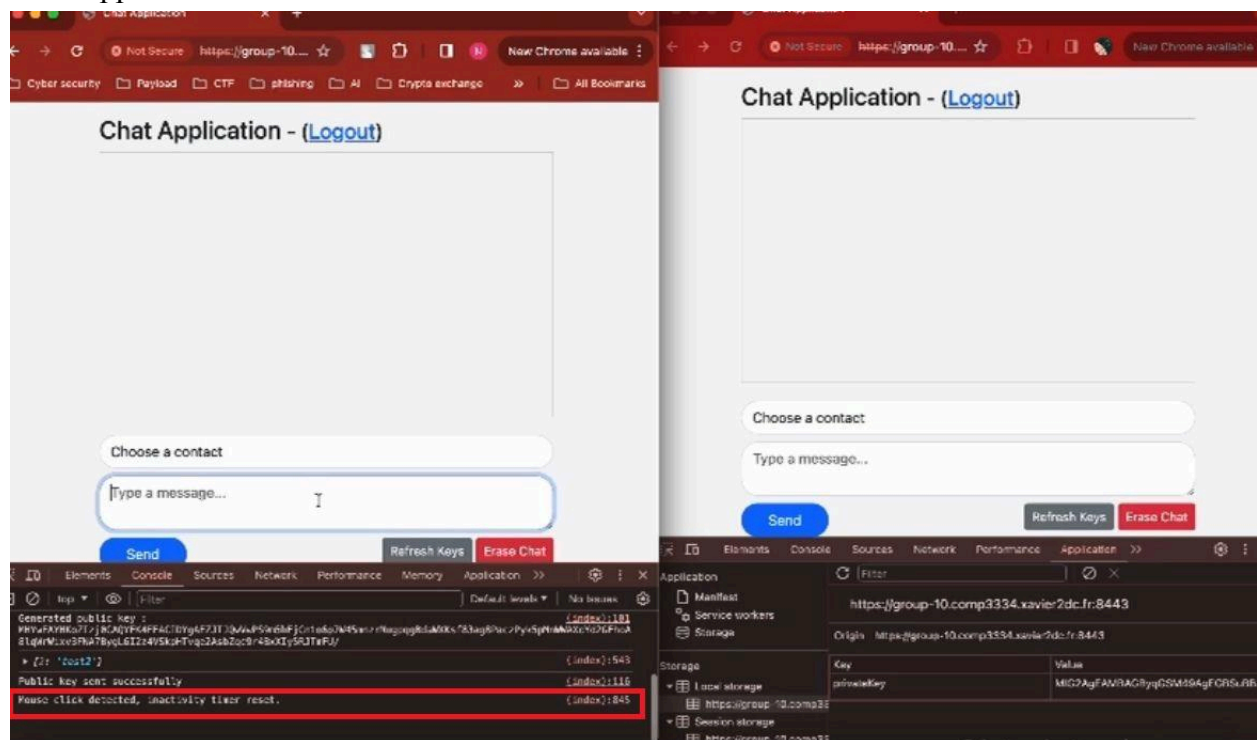
Memorized Secret:

Buttons:



4.1.6. Session Timeout

This measure is done to prevent unauthorized access if a user leaves their device unattended or forgets to log out explicitly. The user will be disconnected from a session after being inactive for 20 minutes. After each user clicks, the timer will be restarted to avoid wrongful disconnections to the application.



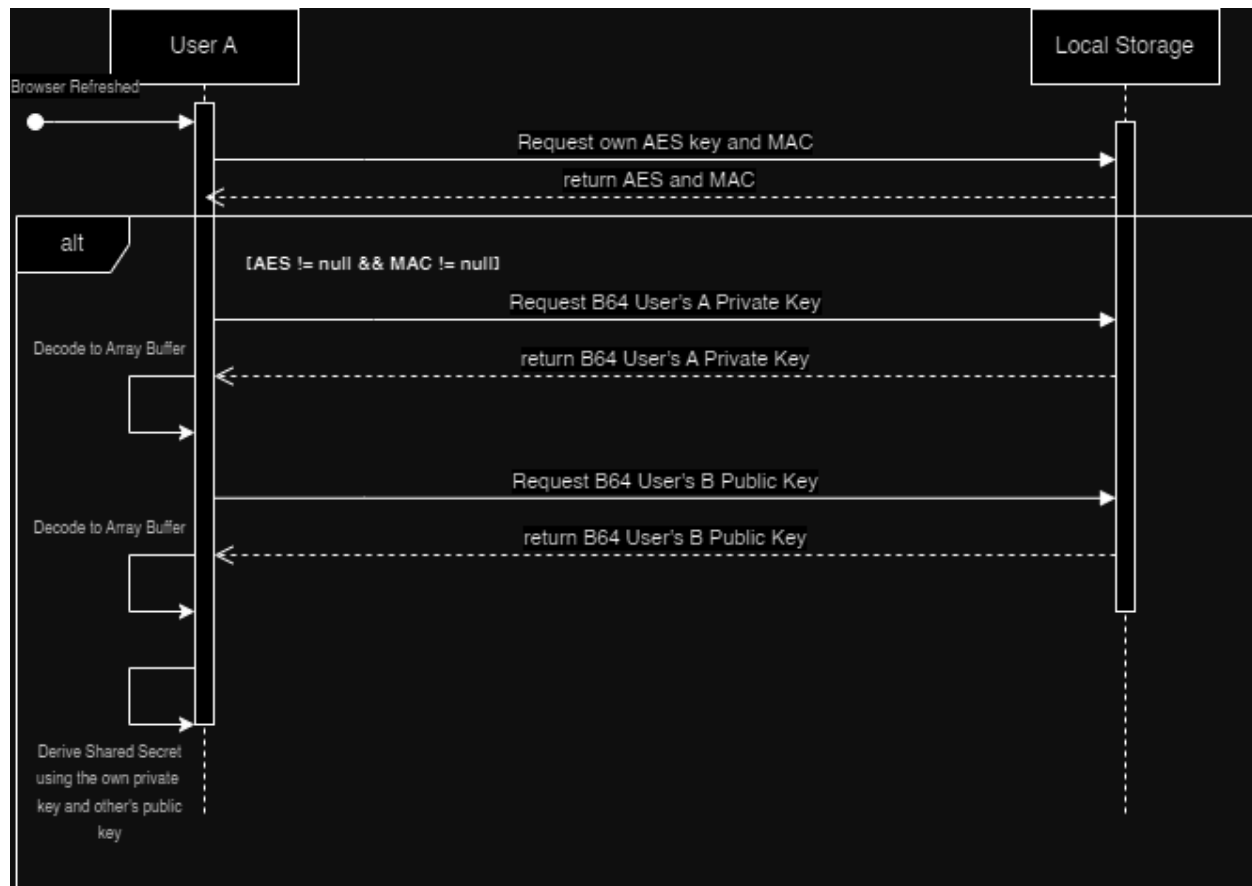
4.2 E2E Encryption

Encryption ensures that only the intended recipient, possessing the corresponding decryption key, can decipher the message content. This approach improves user privacy and ensures confidentiality. Below is a breakdown of our implementation of E2E Encryption.

4.2.1. ECDH Exchange

We use the ECDH key exchange mechanism to allow the two parties to establish a shared secret without revealing the secret itself.

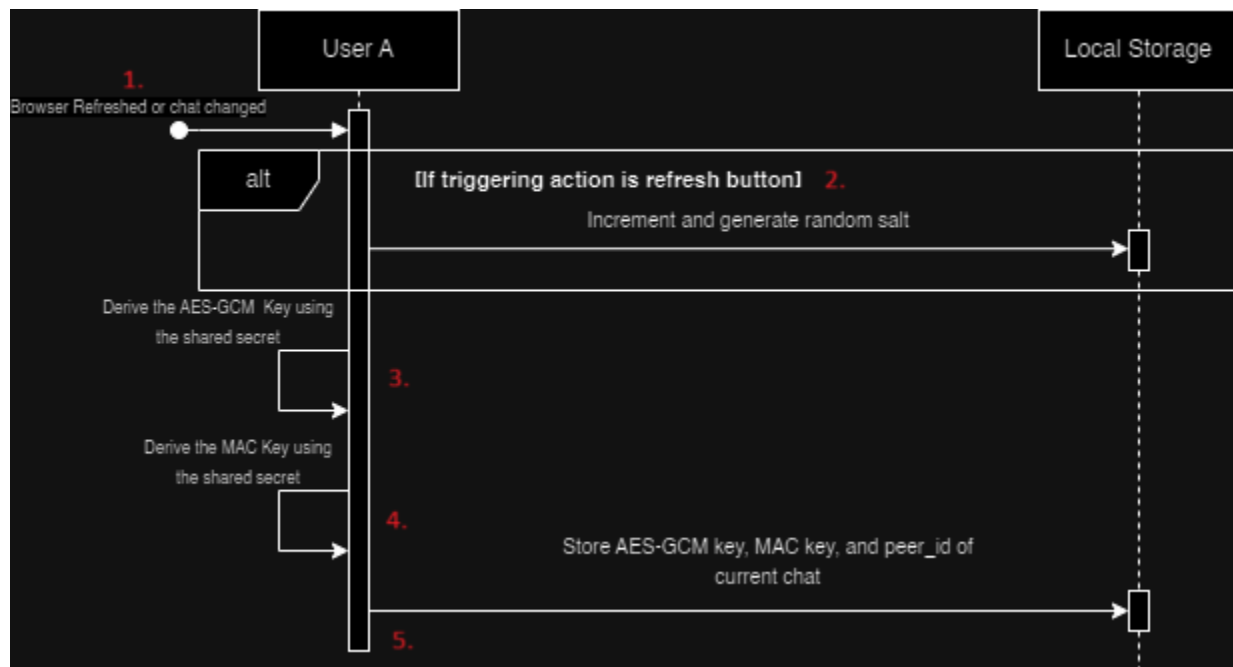
1. Every time user A refreshes the browser (actions such as selecting a different chat would trigger this event), the browser will check if user A has the AES key and MAC for that chat.
2. If user A does not have any of these, a new set of AES key and MAC will be generated
3. User A requests their private key to local storage
4. User A requests user B's public key from the database
5. User A derives the shared secret



4.2.2. Deriving key MAC from the shared secret using

We use HKDF-SHA256 to derive the AES and MAC keys from the shared secret established through ECDH:

1. Derive the keys if one of the following events occurs:
 - a. The refresh button is pressed
 - b. The user changes the current chat
2. If the refresh button was pressed, update the salt
3. Derive the AES-GCM key from the shared secret
4. Derive the MAC key from the shared secret
5. Store the AES-GCM key, MAC key, and the ID of the other user whose current chat is open.



In practice, when the chat is changed:

```
peer changed: 1
Public Key: MHywEAYHkoZiIzj0CAQYFKAEACIDYgAE2hDk601Tto1IGWp1k6oNHMgkw7CZFAHwQ1gFvb9Q9oF9hs648F1XgHTVmhpgwbMzG15FzX/Begy1GCh0Xud1DywdqVH1kkFOAZTC1kr6spzChYw7QnTw1d6ha8p40
no aeskey/mackey found in localstorage, regenerate.
Shared secret: N1Cxcrc4jCI5xLGIpFCjhwA59Hq3ZtW0wUN/A15pxk4-
AAAAAAAAAAAAAAAAAAAAAA--
key info generated: CHAT_KEY_USER1to3
mac info generated: CHAT_MAC_USER1to3
Derived AES Key: ▶ CryptoKey
Derived MAC Key: ▶ CryptoKey
```

4.2.3. Message Format, Encryption and Sending

Messages are encrypted using AES-GCM using the keys derived in the previous steps. Each iv is updated as a counter with each message that uses the same key. The message is within a JSON object before transmission, containing the id of the receiver, the ciphertext, the iv, and the hmac.

```

ivbytes regenerate: AAAAAAAAAAAAAA
Encryption: iv= AAAAAAAAAAAAAA ,hmac= 414254f117b3c4eb42773ce775cbf1bf9800f121bed30ca6447c25a42d6279a2 ,ciphertext= u8Uw+5XUC7sua!pbphZafGyofOb ,associatedData= CHAT_MSG_FROM_3_TO_1
Encrypted message and HMAC signature sent to server

```

Messages sent are encrypted and stored in the database as shown below:



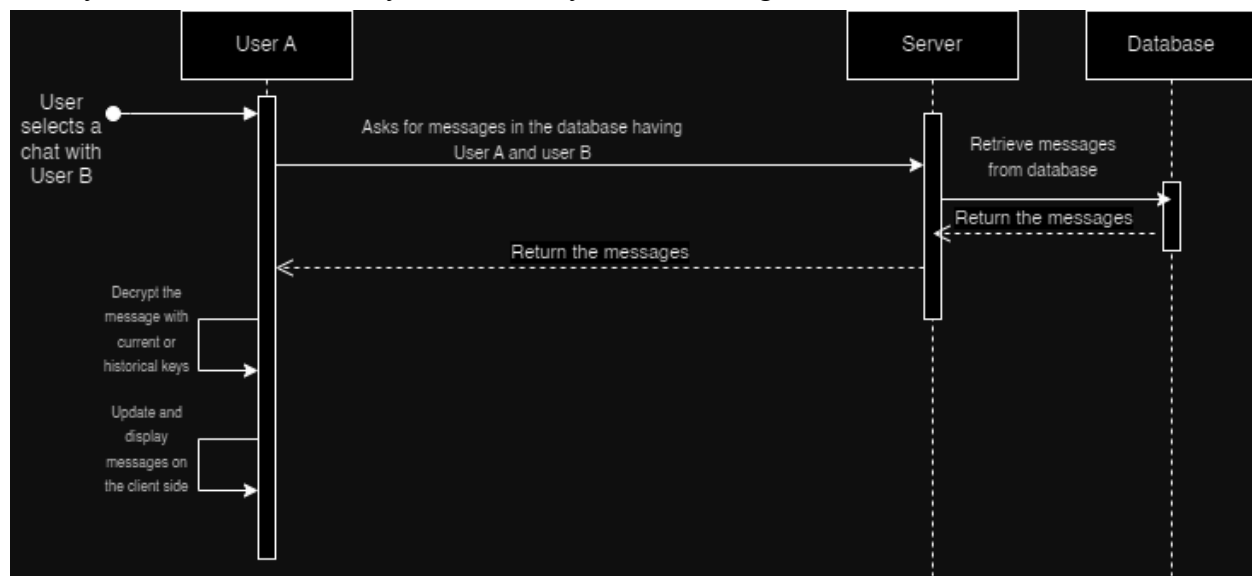
Each message depends on a sender and received user id and its own message id. Messages are stored in the messages table as shown below:

messages	
PK	<u>message_id</u> INT AUTO INCREMENT
	sender_id INT NOT NULL
	receiver_id INT NOT NULL
	ciphertext TEXT NOT NULL, -
	iv TEXT NOT NULL, -- Initialization Vector for AES GCM mod
	hmac TEXT NOT NULL, -- HMAC signature to ensure messa
	aad TEXT NOT NULL, -- Additional Authenticated Data
	created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
	FOREIGN KEY (sender_id) REFERENCES users(user_id)
	FOREIGN KEY (receiver_id) REFERENCES users(user_id)

4.2.4. Loading Previous and New Messages

When users log in again, they expect to encounter their previous chats and receive new messages from when they were not online. The client will contact the server to retrieve the previously encrypted and stored messages from the database.

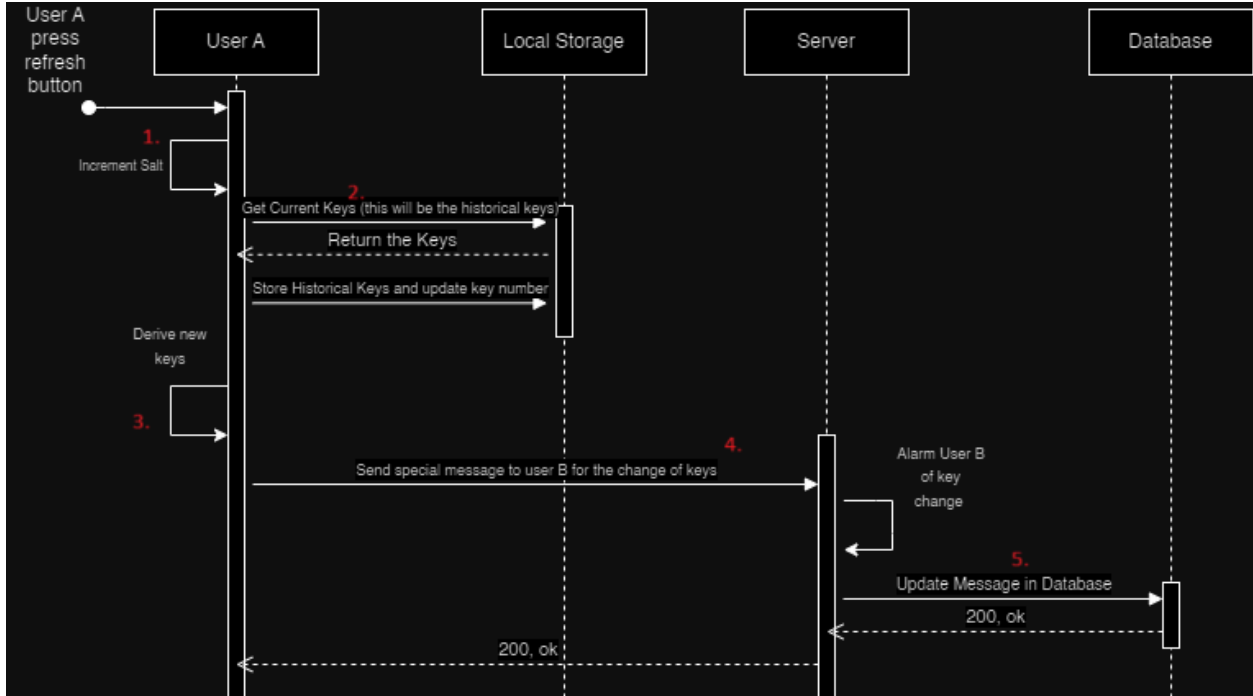
The following sequence diagram assumes the previous process of AES and MAC keys have already been done, and the keys are currently in local storage:



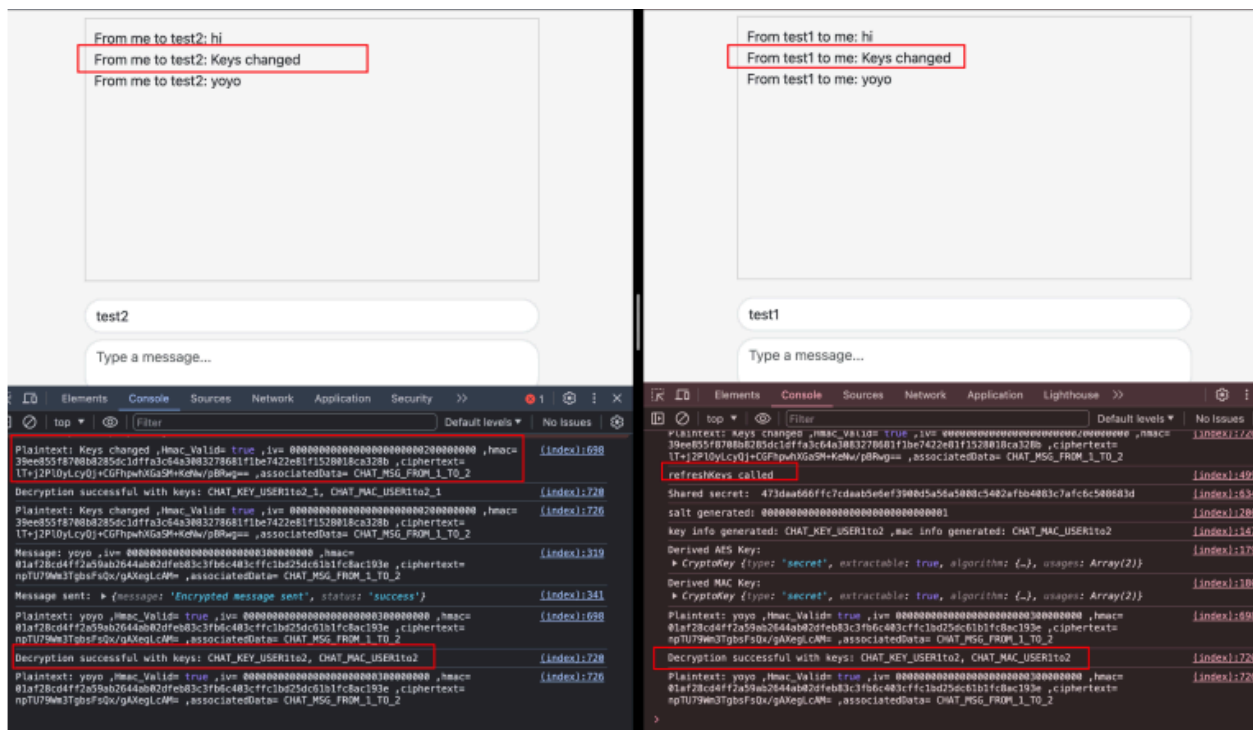
4.2.5. Refresh

When a user clicks the refresh button, new symmetric keys and IVs will be rederived from the same shared secret. To be able to decrypt older messages encrypted with previous keys, we store the older keys, so-called “historical keys”. Salt is incremented, and new keys are rederived with the new salt in the following steps:

1. Salt is incremented and regenerated randomly.
2. The current AES and MAC keys from that chat are gathered and stored in local storage as “historical keys”.
3. User A derives the new keys
4. User A sends a message to User B saying, “Keys changed”, which also works as an alert for User B to be aware of the change of keys.
5. The special message is stored in the database.

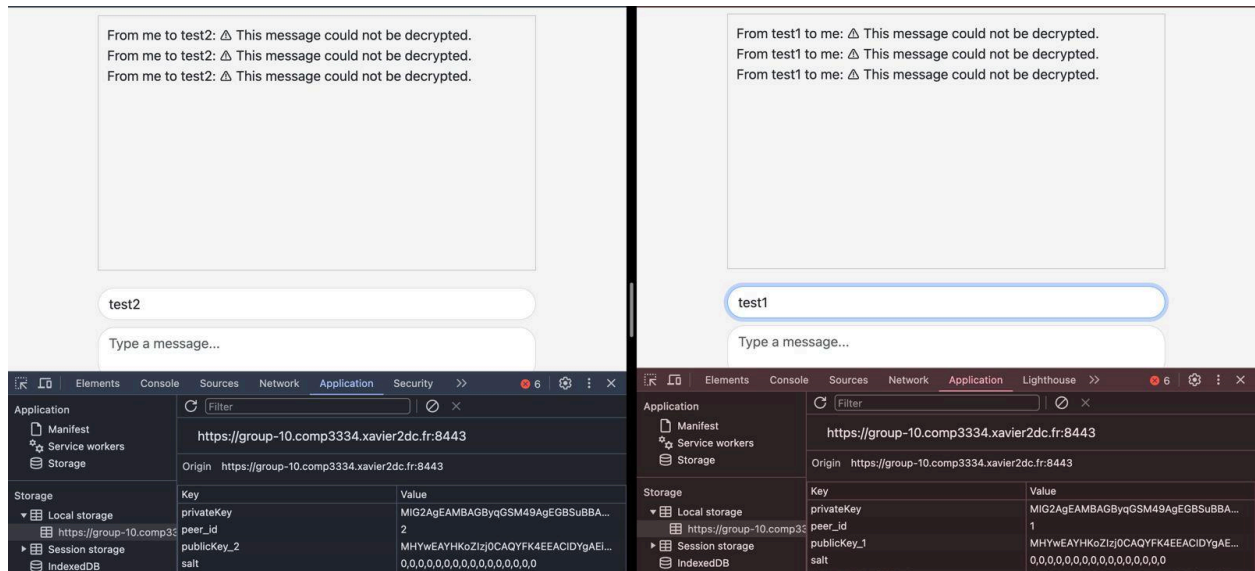


In the web app, this can be observed such as:



If local storage is deleted, previous messages will not be able to be decrypted

In our application, this situation may occur when logout is done, which deletes and cleans the local storage.



4.2.6. SQL Injections, XSS, and CSFR

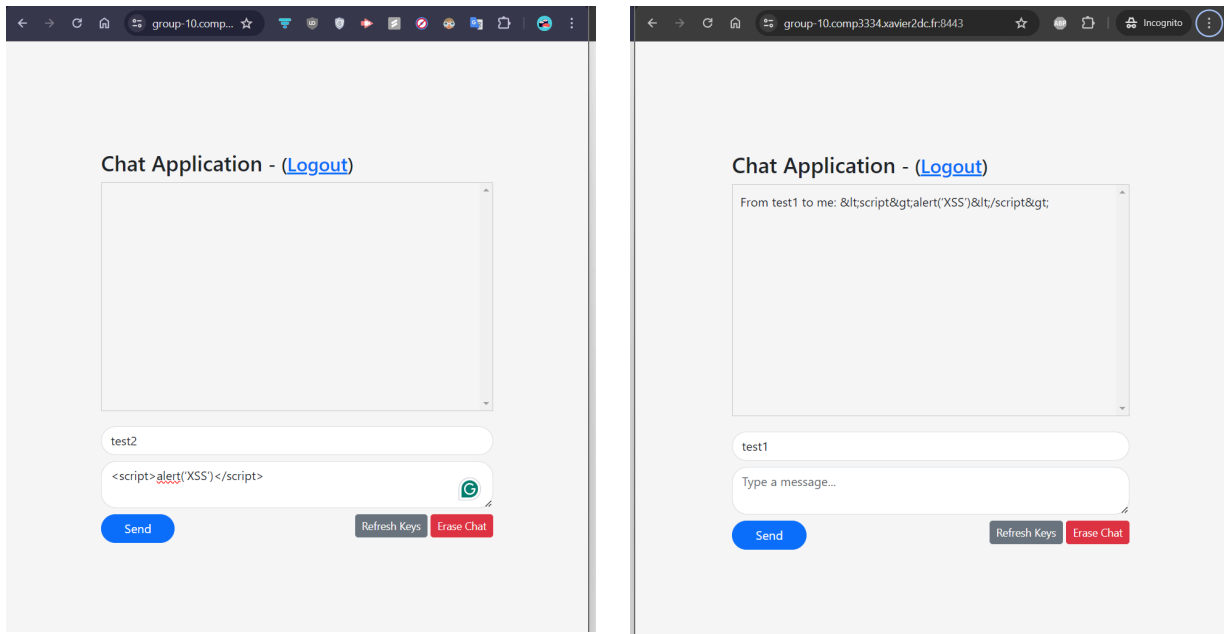
We implemented different countermeasures to mitigate these previously mentioned in our threat model.

Input Validation:

Both **server-side** and **client-side** input validation were handled to ensure more optimal security and correct use of our chat application.

To fight the following threats:

- **SQL Injection (SQLi) Mitigation:** We used prepared statements to separate data from the actual SQL statement. This prevents SQL injection attacks by ensuring data is treated purely as values and not executable code.
- **Cross-Site Scripting (XSS) Mitigation:** The HttpOnly and secure flags are set on cookies to prevent them from being accessed by client-side JavaScript. This enhances security by making it more difficult for XSS attacks to steal sensitive information stored in cookies.



- **Cross-Site Request Forgery (CSRF) Mitigation:** Implementing SameSite=Strict in cookies. In this way, cookies are only sent along with requests initiated from the same website that created the cookie, protecting against CSRF attacks.

4.3 TLS

TLS works as the most extensively used security protocol in the world and the industry standard for protecting browser-server traffic [2]. To ensure TLS in our application, we modified the nginx file to ensure our security requirements and issued our domain certificate from the given CA certificate and private key.

4.3.1. Nginx Configuration File

We configured Nginx to serve our website securely over HTTPS, specifying the paths to our SSL certificate and private key, defining the cipher suite for the TLS handshake, defining the preferred ciphers during the TLS handshake, and defining the elliptic curve for key exchange. Below is a specific implementation of our nginx configuration file

```

events{}

http {
    upstream flask_app {
        server webapp:5000; # Assuming 'webapp' is the service name in docker-compose.yml
    }

    server {
        listen 8080;
        server_name group-10.comp3334.xavier2dc.fr;

        location / {
            proxy_pass http://flask_app;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }

    server {
        listen 8443 ssl;
        server_name group-10.comp3334.xavier2dc.fr;

        # SSL configuration
        ssl_certificate /etc/nginx/group10.crt;
        ssl_certificate_key /etc/nginx/group10.key;
        ssl_protocols TLSv1.3; 1. TLS 1.3 only
        ssl_ciphers ECDHE-ECDSA-AES256-GCM-SHA384;
        ssl_prefer_server_ciphers on;
        ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256; 3. Cipher suite only
        ssl_ecdh_curve X25519; 2. X25519 EC
        ssl_stapling off; 4. No OCSP stapling

        # HSTS configuration (Max age: 1 week)
        add_header Strict-Transport-Security "max-age=604800" always; 5. HSTS for one week

        location / {
            proxy_pass http://flask_app;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}

```

Nginx configuration File

1. TLS version: use only TLS version 1.3. Older versions are disabled.

2. Elliptic Curve: X25519 elliptic curve will be used for key exchange within the chosen ciphersuite.

3. Ciphersuite: ChaCha stream cipher and Poly1305 authenticator with secure hash algorithm SHA256.

4. OSCP Stapling: We do not use OSCP stapling since we are using a self-signed CA certificate.

5. HSTS: Send Strict-Transport-Security to the browser and set the time that the browser should access the site only using HTTPS to one week (604800 seconds).

4.3.2 Domain Certificate

Steps for generating our domain certificate

1. Generating a new private key for our domain certificate

```
./openssl ecparam -out ../../group10.key -name prime256v1 -genkey
```

2. Creating the Certificate Signing Request (CSR)

```

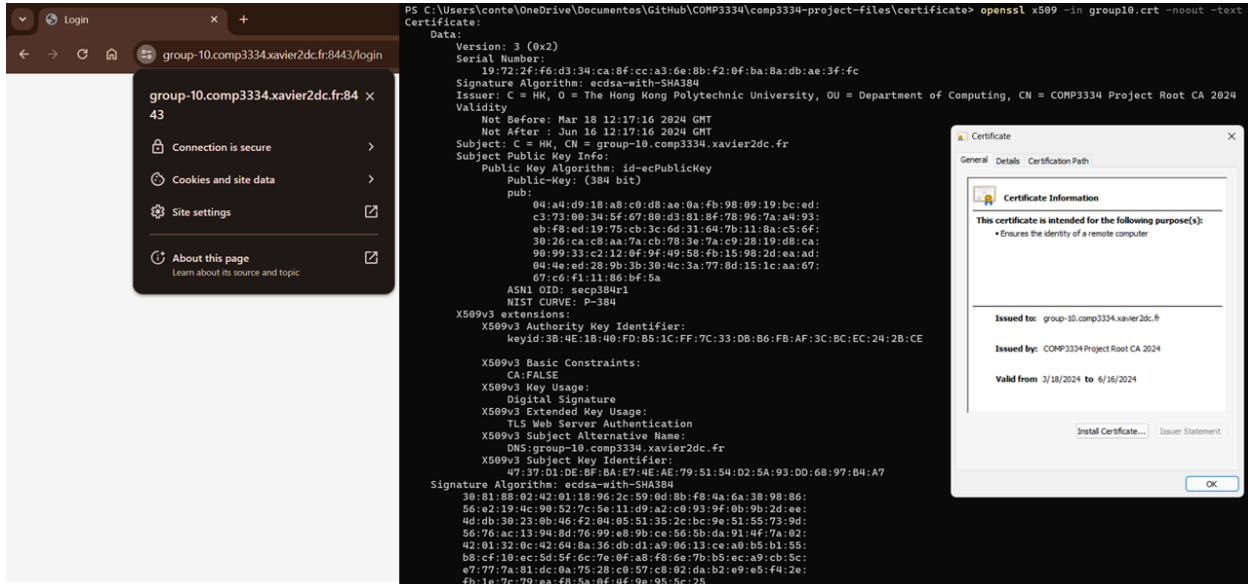
/openssl req -new -key ../../group10.key -out ../../group10.csr -config ../../openssl.cnf
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [HK]:HK
State or Province Name (full name) [Kowloon]:The Hong Kong Polytechnic University
Locality Name (eg, city) []:Hong Kong
Organization Name (eg, company) [PolyU]:Department of Computing
Organizational Unit Name (eg, section) [COMP3334]:COMP3334
Common Name (e.g. server FQDN or YOUR name) []:group-10.comp3334.xavier2dc.fr
Email Address []:

```

3. Sign the CSR with the given CA and CAkey

```
/openssl x509 -req -in ../group10.csr -CA ../cacert.crt -CAkey ../cakey.pem -CAcreateserial -out ../group10.crt -days 365
Certificate request self-signature ok
subject=C=HK, ST=The Hong Kong Polytechnic University, L=Hong Kong, O=Department of Computing, OU=COMP3334, CN=group-10.comp334.xavier2dc.fr
```

Hence, we read and display the domain certificate obtained:



5. Summary

To conclude, we developed a secure end-to-end encrypted chat web application where users exchange messages that only each user can read with their own credentials and keys.

Authentication with memorized secrets and OTP was implemented to ensure the security of the session, key exchange with ECDH, encryption with AES-GCM was implemented for the confidentiality of the messages, TLS was used for securing communications, and a valid certificate for our web application was issued to authenticate our website.

References

- [1] E. T. Lodderstedt, "RFC 6819 - OAuth 2.0 Threat Model and Security Considerations," *datatracker.ietf.org*, Jan. 2013. <https://datatracker.ietf.org/doc/html/rfc6819>
- [2] P. C. Van Oorschot, *Computer Security and the Internet : Tools and Jewels*. Cham: Springer, 2020.