Department of Computing
電子計算學系

THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

COMP3334 Computer System Security

Group 14

End-to-end Encrypted Chat Web Application

Project Report

JIANG Guanlin (21093962d)

YE Feng (21098249d)

HU Yuhang (21106395d)

CAI Haoyu (21103839d)

LUO Yi (21108269d)

Table of Contents

1. INTRODUCTION..... 3
2. PROJECT AIMS AND SCOPE..... 4
2.1 Project Aims..... 4
2.1.1 User-Friendly Interface Design..... 4
2.1.2 Security Protocols and Data Encryption..... 5
3. SYSTEM ANALYSIS..... 6
3.1 System requirements..... 6
3.1.1 Authentication..... 6
3.1.2 E2EE Chat..... 7
3.1.3 TLS..... 7
4. SYSTEM DESIGN..... 8
4.1 Database Design..... 8
4.1.1 Users Table..... 8
4.1.2 Public Keys Exchange Table..... 9
4.1.3 Salt Exchange Table..... 9
4.1.4 Messages Table..... 10
4.1.5 Captchas Table..... 10
4.2 Sequence Diagram..... 10
5. IMPLEMENTATION..... 11
5.1 Security & Encryption Implementation..... 11
5.1.1 Data Encryption & Login Mechanism..... 11
● Password Storage..... 11
● TOTP..... 11
● Captcha..... 12
5.1.2 Message Encryption..... 12
● Keys Generation..... 12
● Message Encryption & HMAC Sign..... 13
● Message Decryption & HMAC Verify..... 13
5.1.3 Session & Cookie..... 14
5.1.4 Prevention of SQL injection & XSS Attack & CSFR Attack..... 14
● Prepared Statements..... 14
● Session Cookies Extension..... 15
5.1.5 HTTPS Protection..... 15
6. CONCLUSIONS..... 15
7. REFERENCES..... 16

1. INTRODUCTION

In today's digital world, privacy, and security are super important, so we've started working on an end-to-end encrypted chat application. Our goal is to make sure that messages between users stay private, and that even the servers can't peek at them. We're using some of the latest encryption methods and really strict security protocols to keep everything safe.

We're using AES encryption in GCM mode which is one of the best choices because it's both strong and fast. Every message gets encrypted so that no one else can read it, and each message comes with a special code to make sure it's unique and hasn't been tampered with. We also use another technique called HMAC-SHA256 to protect these codes from being messed with.

In order to make things still work even if users close their browsers, we use HTML5 Local Storage. This helps us keep users' key information safe but ready for when they log back in. The refreshing key, which means updating the encryption keys and MAC keys manually, can keep the message secure.

Furthermore, we're also careful about how encryption keys are swapped between users. Our system handles all the details, making sure everything's secure from start to finish. We put a lot of work into preventing unauthorized access and ensuring no one can alter any part of the messaging process.

Besides the encryption inside the chat, we also protect messages while they're traveling from one place to another using TLS, which is a standard security technology for keeping an internet connection secure. We make sure to use the latest version and follow strict guidelines to keep our communications safe. From cross-site request forgery (CSRF) to cross-site scripting (XSS) and SQL injection attacks, our system stands as an impregnable fortress, safeguarding user data against all forms of malicious exploitation.

In the end, an end-to-end encrypted chat application is a simple and powerful application that puts privacy and security in the first place. Our users will feel confident and secure in their digital conversations, knowing that this application can always protect their privacy with the latest and greatest technology.

2. PROJECT AIMS AND SCOPE

2.1 Project Aims

Our project aims to develop a cutting-edge end-to-end encrypted (E2EE) chat application that prioritizes user privacy and security without compromising on usability. By leveraging advanced cryptographic techniques and adhering to stringent security protocols, our goal is to create a platform where users can engage in confidential conversations with the assurance that their messages remain inaccessible to any unauthorized parties, including the server itself. Through meticulous attention to detail and innovative design, we seek to redefine the standards of secure communication in the digital age, empowering users with a seamless yet highly secure means of exchanging information.

2.1.1 User-Friendly Interface Design

Our interface design prioritizes ease of use and accessibility. It features clear navigation, consistent design elements, and interactive feedback for a seamless user experience. Accessibility features are integrated to cater to diverse user needs, while user-centric functionality ensures continuous improvement based on feedback.

2.1.2 Security Protocols and Data Encryption

In our pursuit of maintaining stringent security standards, we have implemented a series of robust security protocols to safeguard user data and interactions. Detailed explanations of these practices are elaborated in Session Five of our implementation.

Data Encryption & Transfer Protection: Our system employs the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) with Initialization Vectors (IV) to ensure the confidentiality and integrity of messages. Additionally, we utilize the Hash-based Message Authentication Code (HMAC) algorithm to hash the IV, thereby fortifying it against unauthorized modifications by potential attackers. Furthermore, we reinforce data transfer protection through Transport Layer Security (TLS/SSL) with the Nginx web server, utilizing self-assigned certification to encrypt data during transit. With the HTTPS layer in place, the secure transfer of public keys generated by the Elliptic Curve Diffie-Hellman (ECDH) algorithm, alongside counter-based salts, enhances the safety of key derivation processes.

User Login Security: For securing user login, we use the Python BCrypt library Hashing to generate the hash for the password and use the Time-based one-time password (TOTP), which uses Google Authenticator to make the two-factor authentication, one of the multi-factor authentication. To generate the TOTP, we use the Python PyOTP library to use AES CBC mode to protect the recovery code with IV. We believe this double approach can provide a robust defense against unauthorized access to passwords, given there might be any malicious admin on the server side, the database is leaked unexpectedly, or the password from the user side leaked. With this implementation, potential dictionary attacks by the adversary are computationally infeasible because of the one-way property of the cryptographic hash function. We also added the image captcha to slow down the speed of the login, if an attacker wants to try login multiple times, the system can slow down, also has the limitation of the login times.

Login Procedure: The login procedure of the E2EE Chat Web APP relies on session tokens to maintain user authentication and authorization. These tokens play a pivotal role in ensuring that only authenticated users gain access to the system's functionalities, thereby bolstering overall security.

3. SYSTEM ANALYSIS

3.1 System requirements

3.1.1 Authentication

- REQ1: **User Authentication** - The system shall provide a login mechanism for users to access the secure chat functionality. Users should be required to authenticate themselves using appropriate credentials.
- REQ2: **Password Hashing** - User-chosen memorized secrets shall be salted and hashed using a suitable one-way key derivation function. The system shall use the NIST-recommended password hashing function called bcrypt for salted hashing.
- REQ3: **Memorized Secret Verification** - The system shall implement functionality to compare user-chosen memorized secrets against previously leaked datasets.
- REQ4: **Multi-Factor Authentication** - The system should use the authenticators which are user-chosen memorized secrets (passwords), single-factor OTP devices (Google

Authenticator), and look-up secrets (recovery keys) to authenticate users and authorize them to access systems or resources.

- REQ5: **Authentication Security Measures** - The system should implement rate-limiting mechanisms to limit the number of authentication attempts per unit of time to prevent brute-force attacks and image-based CAPTCHAs to ensure that users are human and not automated programs in the user registration and login interface.
- REQ6: **Session Security** - After successful authentication, the system should ensure that sessions are bound to the user's identity to prevent session hijacking and identity spoofing attacks.
- REQ7: **Authenticator Binding** - During new account registration, the system should bind authenticators (OTP and recovery keys) simultaneously. Provide options during the registration process to set up and bind all required authenticators.
- REQ8: **OTP Authenticator Cloning Prevention** - For OTP authenticators, particularly software-based OTP generators, the system should discourage and prevent the cloning of secret keys onto multiple devices. Clearly state in the documentation and settings of OTP authenticators that key cloning onto multiple devices is not encouraged and provide security recommendations to prevent key leakage and misuse.

3.1.2 E2EE Chat

- REQ1: **ECDH-based shared secret** - The system should implement the ECDH key exchange protocol to establish a shared secret between the users.
- REQ2: **Key Derivation** - The system should derive two 256-bit AES-GCM encryption keys and two 256-bit MAC keys from the shared secret using HKDF-SHA256.
- REQ3: **Message Encryption** - Chat messages should be encrypted using AES in GCM mode.
- REQ4: **Key Storage** - All key material should be stored in the HTML5 Local Storage of the user's browser to be retrieved after the browser is reopened.
- REQ5: **Message History** - The system should display the history of previous messages exchanged and new messages.
- REQ6: **Key Refresh** - The system should allow users to refresh their keys by re-deriving all symmetric keys and IVs from the shared secret.

- REQ7: **Key Exchange Initialization** - When the Local Storage is cleared or there is no shared secret for a given recipient, the sender should initiate the ECDH key exchange using a special message.
- REQ8: **Message Encoding and Format** - Chat messages should be encoded using UTF-8. Network messages between users should be formatted in JSON using a custom schema.
- REQ9: **Logging and Visual Indicators** - The system should log all cryptographic operations including key, IV, plaintext and etc. by using console.log().
- REQ10: **Security Measures** - The chat app should be protected against cross-site request forgery (CSRF), cross-site scripting (XSS), and SQL injection attacks.

3.1.3 TLS

- REQ1: **Version** - TLS version shall be 1.3 only.
- REQ2: **Key Exchange** - The system shall use x25519 Elliptic Curve Group only for key exchange during the handshake process.
- REQ3: **Cryptographic Algorithms** - The system shall use the TLS_CHACHA20_POLY1305_SHA256 cipher suite only for securing the communication between a client and a server.
- REQ4: **Self-signed CA Certificate** - The system should not use OCSP stapling to enhance the efficiency and security of the OCSP for checking the revocation status of digital certificates.
- REQ5: **Validity Period** - The validity period of HSTS which is to enforce secure connections over HTTPS should be one week.
- REQ6: **TLS Certificate** - Its version should be X.509 version 3. It includes a public key that is generated using the ECDSA algorithm with the P-384 elliptic curve. Its hashing algorithm for signature is SHA384. Its CA flag (critical) is false, key usage (critical) is a digital signature, extended key usage is server authentication, and the validity period is 90 days. It Includes both the Subject Key Identifier and the Authority Key Identifier.
- REQ7: **Website** - The website is hosted at https://group-14.comp3334.xavier2dc.fr:8443/
- REQ8: **Redirection** - The system should make the website redirect to 127.0.0.1.

- REQ9: **Issue of Certificate** - The system should issue the certificate from the given CA certificate and private key.
- REQ10: **Safety of CA Certificate** - The CA certificate should be domain-constrained to subdomains of comp3334.xavier2dc.fr, meaning you can safely trust it on your computer (nobody can generate valid certificates for other domains)

4. SYSTEM DESIGN

4.1 Database Design

4.1.1 Users Table

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    mfa_secret VARCHAR(255),
    mfa_enabled BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

For the users table:

- User_id as the primary key;
- Username, password, mfa_secret, mfa_enabled, createed_at;

Note that as we consider the password and mfa_secret as sensitive data, according to the Principle of adequate protection, we encrypt mfa_secret and hash the password to store those encrypted data in the database.

4.1.2 Public Keys Exchange Table

```
CREATE TABLE public_keys_exchange (
    public_key_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    public_key TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

For the public keys exchange table:

- Public_key_id as the primary key;
- user_id , public_key, created_at;

This table is for the public key store, and public key fetch, when the user selects the peer, the peer public key will be fetched. Also, when the user login to the system, if the local does not have the key, the key pair also will be generated and send the public key to the database.

4.1.3 Salt Exchange Table

```
CREATE TABLE salt_exchange (
    salt_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    salt TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

For the salt exchange table:

- Salt_id as the primary key;
- user_id, salt, created_at;

This table is for the salt store, and salt fetch, when the user selects the peer, the peer salt will be fetched. Also, when the user login to the system, if the local does not have the salt, the salt also will be generated and sent the salt to the database.

4.1.4 Messages Table

```
CREATE TABLE messages (
    message_id INT AUTO_INCREMENT PRIMARY KEY,
    sender_id INT NOT NULL,
    receiver_id INT NOT NULL,
    message_text TEXT NOT NULL,
    iv TEXT NOT NULL,
    hmac TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (sender_id) REFERENCES users(user_id),
    FOREIGN KEY (receiver_id) REFERENCES users(user_id)
);
```

For the messages table:

- Messages_id as the primary key;
- Sender_id, receiver_id, message_text, iv, hmac, created_at;

This table is for storing the messages, which are already encrypted, and also saves the iv and hmac of iv, which more easy for peers to verify the integrity of the iv.

4.1.5 Captchas Table

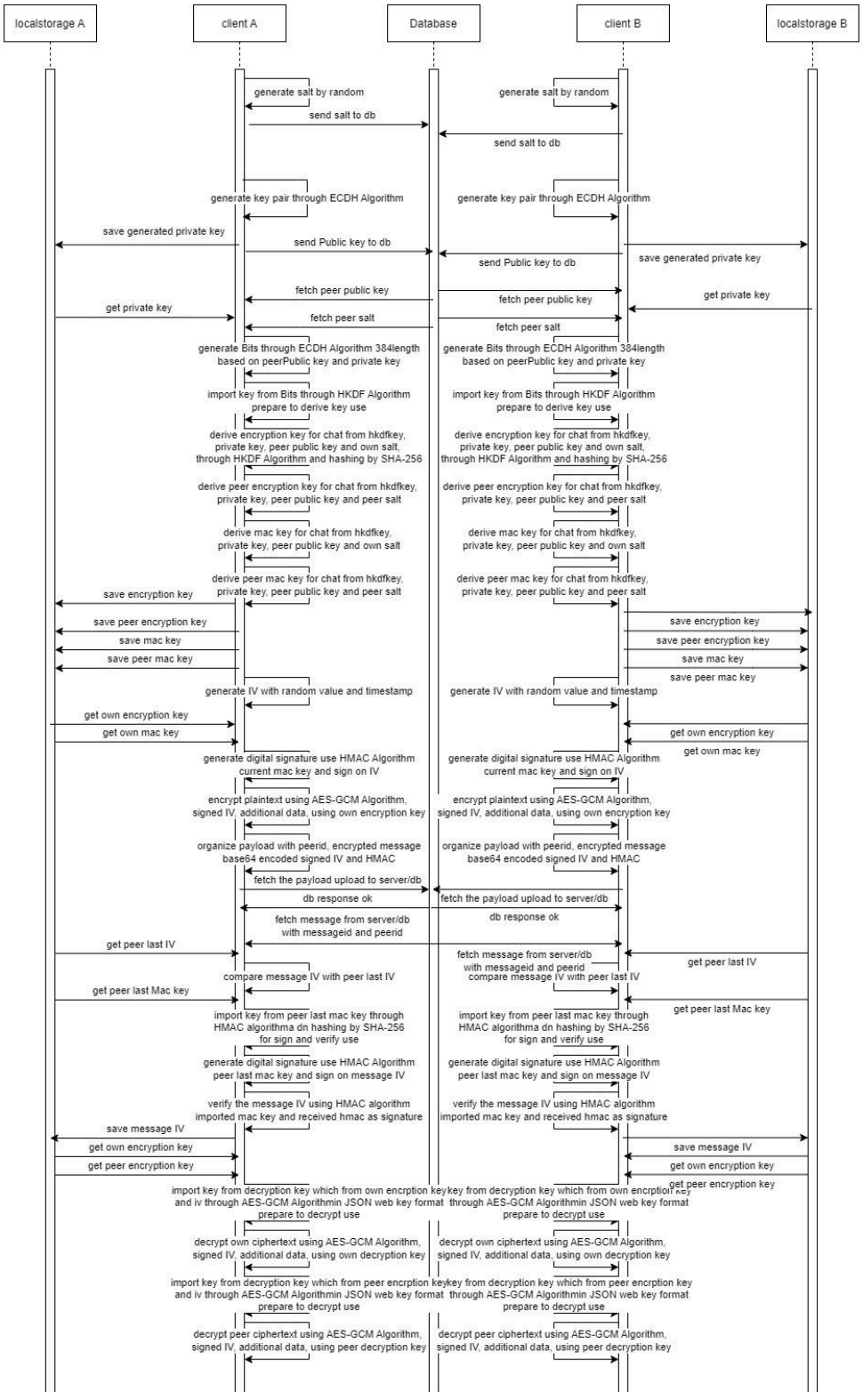
```
CREATE TABLE captchas (
  id INT AUTO_INCREMENT PRIMARY KEY,
  captcha_id VARCHAR(10) NOT NULL,
  captcha_text VARCHAR(6) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

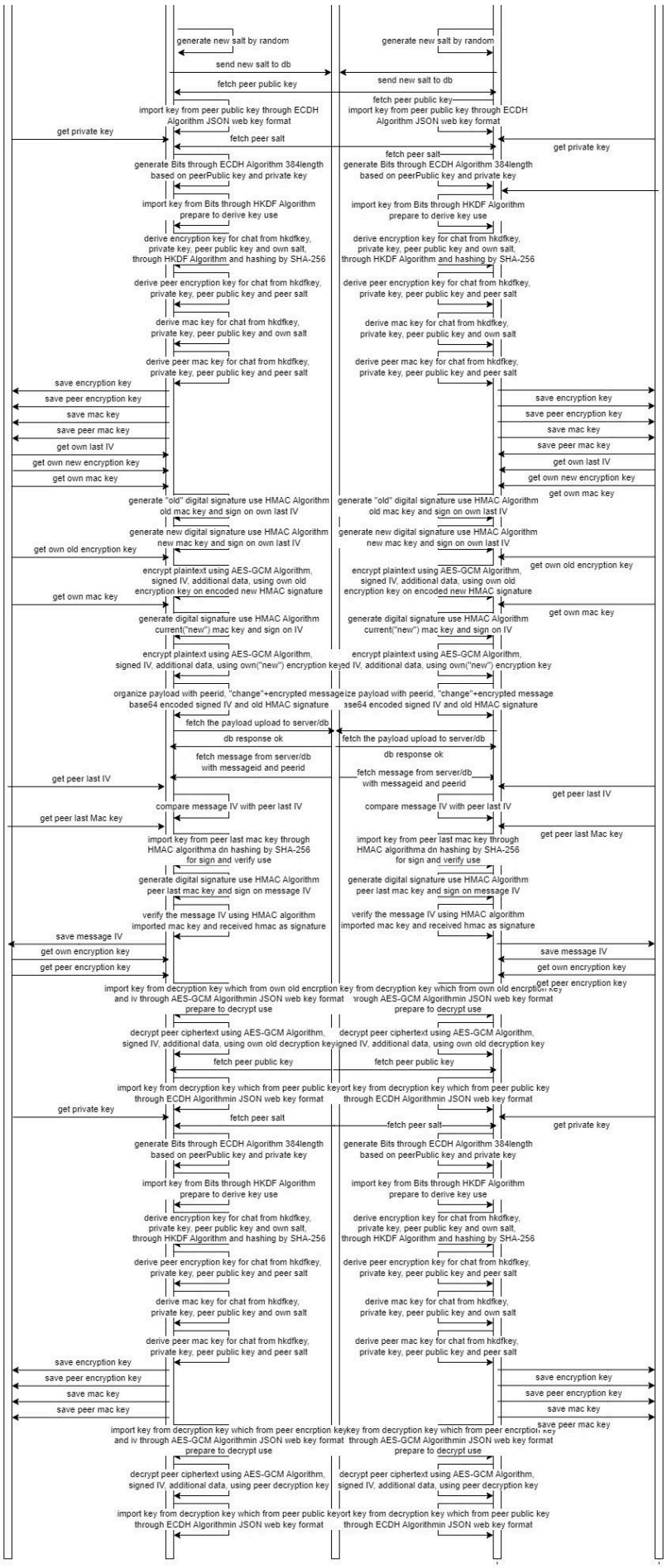
For the captchas table:

- Id as the primary key;
- captcha_id, captcha_text, created_at;

This table is for storing the captchas and id to help verify the captchas that user input, for some seconds the captcha will be deleted and changed.

4.2 Sequence Diagram




generate new salt by random
generate new salt by random
send new salt to db
send new salt to db
fetch peer public key
fetch peer public key
import key from peer public key through ECDH Algorithm JSON web key format
import key from peer public key through ECDH Algorithm JSON web key format
get private key
fetch peer salt
get private key
fetch peer salt
generate Bits through ECDH Algorithm 384length based on peerPublic key and private key
generate Bits through ECDH Algorithm 384length based on peerPublic key and private key
import key from Bits through HKDF Algorithm prepare to derive key use
import key from Bits through HKDF Algorithm prepare to derive key use
derive encryption key for chat from hkdfkey, private key, peer public key and own salt, through HKDF Algorithm and hashing by SHA-256
derive encryption key for chat from hkdfkey, private key, peer public key and own salt, through HKDF Algorithm and hashing by SHA-256
derive peer encryption key for chat from hkdfkey, private key, peer public key and peer salt
derive peer encryption key for chat from hkdfkey, private key, peer public key and peer salt
derive mac key for chat from hkdfkey, private key, peer public key and own salt
derive mac key for chat from hkdfkey, private key, peer public key and own salt
derive peer mac key for chat from hkdfkey, private key, peer public key and peer salt
derive peer mac key for chat from hkdfkey, private key, peer public key and peer salt
save encryption key
save peer encryption key
save mac key
save peer mac key
get own last IV
get own new encryption key
get own mac key
save encryption key
save peer encryption key
save mac key
save peer mac key
get own last IV
get own new encryption key
get own mac key
generate "old" digital signature use HMAC Algorithm old mac key and sign on own last IV
generate "old" digital signature use HMAC Algorithm old mac key and sign on own last IV
generate new digital signature use HMAC Algorithm new mac key and sign on own last IV
generate new digital signature use HMAC Algorithm new mac key and sign on own last IV
get own old encryption key
encrypt plaintext using AES-GCM Algorithm, signed IV, additional data, using own old encryption key on encoded new HMAC signature
encrypt plaintext using AES-GCM Algorithm, signed IV, additional data, using own old encryption key on encoded new HMAC signature
get own old encryption key
get own mac key
generate digital signature use HMAC Algorithm current("new") mac key and sign on IV
generate digital signature use HMAC Algorithm current("new") mac key and sign on IV
get own mac key
encrypt plaintext using AES-GCM Algorithm, signed IV, additional data, using own("new") encryption key
encrypt plaintext using AES-GCM Algorithm, signed IV, additional data, using own("new") encryption key
organize payload with peerid, "change"+encrypted message base64 encoded signed IV and old HMAC signature
organize payload with peerid, "change"+encrypted message base64 encoded signed IV and old HMAC signature
fetch the payload upload to server/db
db response ok
fetch the payload upload to server/db
db response ok
fetch message from server/db with messageid and peerid
fetch message from server/db with messageid and peerid
get peer last IV
get peer last IV
compare message IV with peer last IV
compare message IV with peer last IV
get peer last Mac key
get peer last Mac key
import key from peer last mac key through HMAC algorithma dn hashing by SHA-256 for sign and verify use
import key from peer last mac key through HMAC algorithma dn hashing by SHA-256 for sign and verify use
generate digital signature use HMAC Algorithm peer last mac key and sign on message IV
generate digital signature use HMAC Algorithm peer last mac key and sign on message IV
verify the message IV using HMAC algorithm imported mac key and received hmac as signature
verify the message IV using HMAC algorithm imported mac key and received hmac as signature
save message IV
get own encryption key
get peer encryption key
save message IV
get own encryption key
get peer encryption key
import key from decryption key which from own old encrption key and iv through AES-GCM Algorithmin JSON web key format prepare to decrypt use
import key from decryption key which from own old encrption key through AES-GCM Algorithmin JSON web key format prepare to decrypt use
decrypt peer ciphertext using AES-GCM Algorithm, signed IV, additional data, using own old decryption key
decrypt peer ciphertext using AES-GCM Algorithm, signed IV, additional data, using own old decryption key
fetch peer public key
fetch peer public key
import key from decryption key which from peer public key through ECDH Algorithmin JSON web key format
import key from decryption key which from peer public key through ECDH Algorithmin JSON web key format
get private key
fetch peer salt
fetch peer salt
get private key
generate Bits through ECDH Algorithm 384length based on peerPublic key and private key
generate Bits through ECDH Algorithm 384length based on peerPublic key and private key
import key from Bits through HKDF Algorithm prepare to derive key use
import key from Bits through HKDF Algorithm prepare to derive key use
derive encryption key for chat from hkdfkey, private key, peer public key and own salt, through HKDF Algorithm and hashing by SHA-256
derive encryption key for chat from hkdfkey, private key, peer public key and own salt, through HKDF Algorithm and hashing by SHA-256
derive peer encryption key for chat from hkdfkey, private key, peer public key and peer salt
derive peer encryption key for chat from hkdfkey, private key, peer public key and peer salt
derive mac key for chat from hkdfkey, private key, peer public key and own salt
derive mac key for chat from hkdfkey, private key, peer public key and own salt
derive peer mac key for chat from hkdfkey, private key, peer public key and peer salt
derive peer mac key for chat from hkdfkey, private key, peer public key and peer salt
save encryption key
save peer encryption key
save mac key
save peer mac key
save encryption key
save peer encryption key
save mac key
save peer mac key
import key from decryption key which from peer encrption key and iv through AES-GCM Algorithmin JSON web key format prepare to decrypt use
import key from decryption key which from peer encrption key through AES-GCM Algorithmin JSON web key format prepare to decrypt use
decrypt peer ciphertext using AES-GCM Algorithm, signed IV, additional data, using peer decryption key
decrypt peer ciphertext using AES-GCM Algorithm, signed IV, additional data, using peer decryption key
import key from decryption key which from peer public key through ECDH Algorithmin JSON web key format
import key from decryption key which from peer public key through ECDH Algorithmin JSON web key format

5. IMPLEMENTATION

5.1 Security & Encryption Implementation

5.1.1 Data Encryption & Login Mechanism

In the End-to-end Encrypted Chat Web Application, ensuring the security of user credentials is of importance.

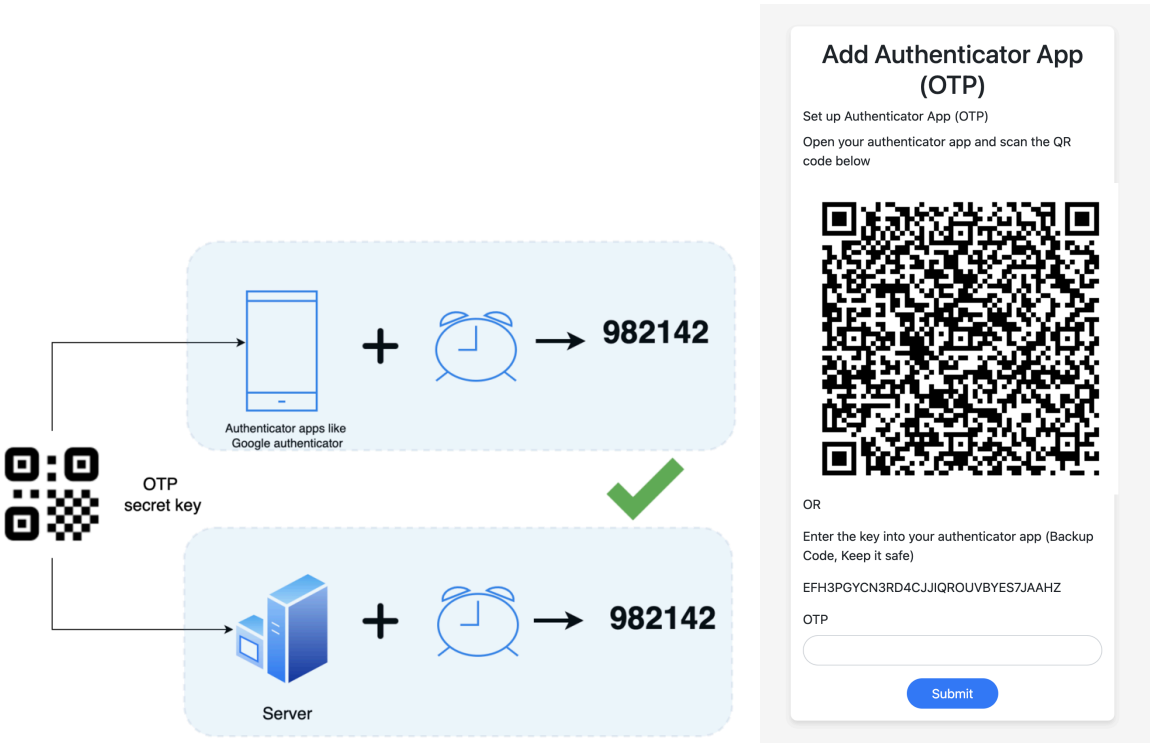
- **Password Storage**

We use the Bcrypt hashing function to hash the user password and store it in the database. Bcrypt can automatically generate the salt and merge the hashing value, salt, and the position of the value and salt together which can prevent the rainbow table attack. Bcrypt library is specialized for password hashing which is based on the Blowfish cipher algorithm to hash the password.

```
def passwordHashing(password):
    return bcrypt.hashpw(password.encode('utf-8'),
bcrypt.gensalt()).decode('utf-8')
```

- **TOTP**

We also use TOTP which is the time-based one-time password to be the multi-factor authentication, users need to use Google Authenticator or Microsoft Authenticator this kind of software to set up. The setup key which is the secret key or recovery key will be given to the user, this key only be given once the user signs up for the system, and the user needs to keep this key safe, and also have the QR Code for the user to scan using the software.



[https://www.linkedin.com/pulse/how-time-based-one-time-password-totp-algorithm-works-vijay-patil]

- **Captcha**

In this application, we add the captcha in the user signup and login steps, which can slow down the speed of user login and signup. In addition, if the attacker wants to implement automatic login through the script, the verification code mechanism will also cause the script to stop at the verification code step. Our verification code also has a time limit. If the user does not enter the verification code within a short period of time, the verification code will become invalid and the verification code needs to be updated before it can be used.

5.1.2 Message Encryption

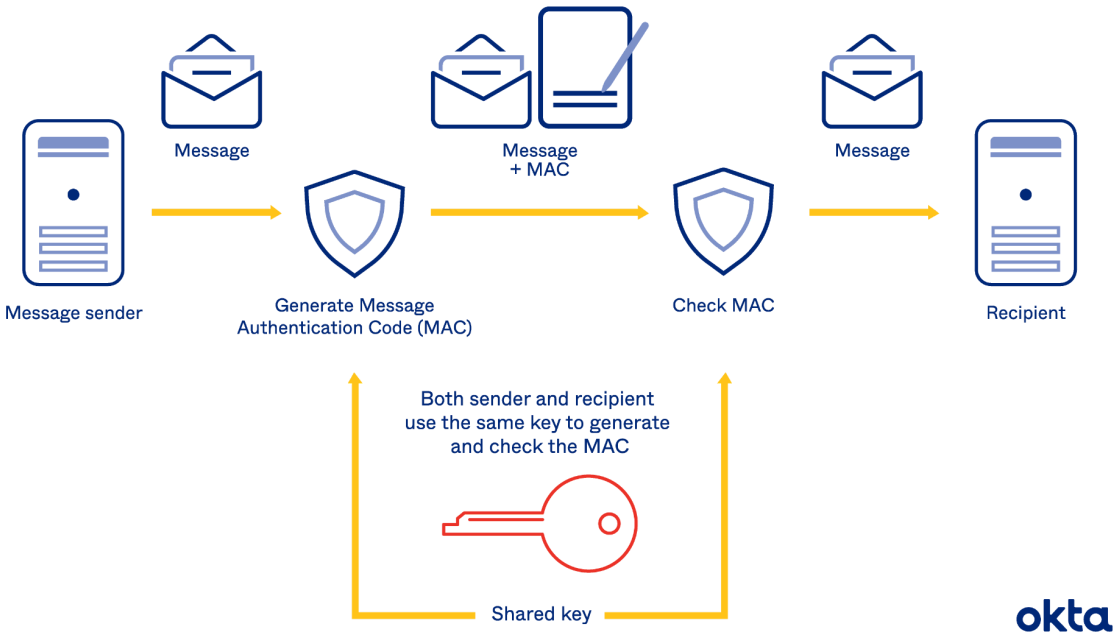
- **Keys Generation**

In the end-to-end encrypted chat web application, chatting is the most important part of this application. When a user logs into the web app, a key pair and salt will be generated under the ECDH with the P-384 algorithm if the local storage is empty and no key is stored, the public key and salt will be sent to the server waiting for another user to select. When two sides get the peer public key and salt, they can use the peer public key and own private key to derive the same

shared secret which is bits format, and import it to the crypto platform which can generate the HKDF key. Using this HKDF key, the info that both sides know and the peer salt which can derive the peer encryption key but uses AES GCM mode for encryption and decryption, and the peer mac key uses HMAC with the hashing function SHA256 for sign and verify. For your own side key generation is the same as generating peer one, but using your own salt and own public key.

- **Message Encryption & HMAC Sign**

When the user sends messages, the message will be encrypted using its own encryption key with AES GCM mode, the IV also will be generated which is based on the timestamp. The IV will also be hashed by HMAC, the HMAC function will use its own mac key, which protects the IV integrity. The HMAC hash, encryption text, and it will be sent to the server waiting for peers to fetch it.



[https://www.okta.com/identity-101/hmac/]

- **Message Decryption & HMAC Verify**

When the peer gets the encryption text, IV, and HMAC hash, the IV needs to be verified, the peer uses the peer mac key, and the HMAC hash is generated by the peer to verify whether the IV is

changed or not. Also, the current IV is bigger than the last IV. When verification is successful, the message will use the peer encryption key to decrypt, and display the message to the user chat box.

5.1.3 Session & Cookie

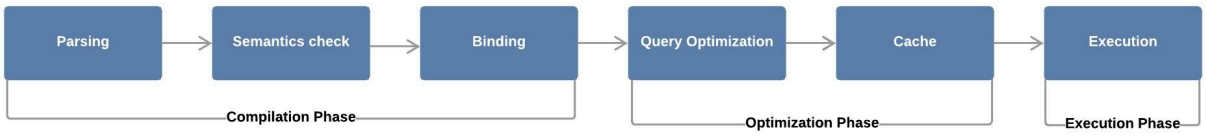
The end-to-end encrypted chat web application uses a session cookie, which is a session token that includes user information such as user ID and user name and a token generated by an encryption algorithm on this data. The session token will be stored in the user's device as the cookie, if the user does not log out and the session cookie has not been deleted when the user into the system at next time, the user is not required to log in again, can just use the system.

5.1.4 Prevention of SQL injection & XSS Attack & CSFR Attack

The End-to-end Encrypted Chat Web Application employs a comprehensive approach to safeguard against SQL Injection (SQLi), XSS, and CSFR attacks, a critical security threat in web applications. This strategy encompasses the use of Prepared Statements (blacklist), and session cookies extensions.

- **Prepared Statements**

  A critical component of the security framework in the E2EE Application is the implementation of prepared statements, a robust technique used to mitigate SQL Injection (SQLi) attacks. This section details how prepared statements are integrated into our system to ensure the protection of sensitive data and the integrity of our database interactions.



  Parameterized Queries: In our platform, SQL queries are constructed with placeholders for user inputs instead of incorporating the inputs directly into the query string. These placeholders are then substituted with actual user inputs in a controlled manner by the database engine.

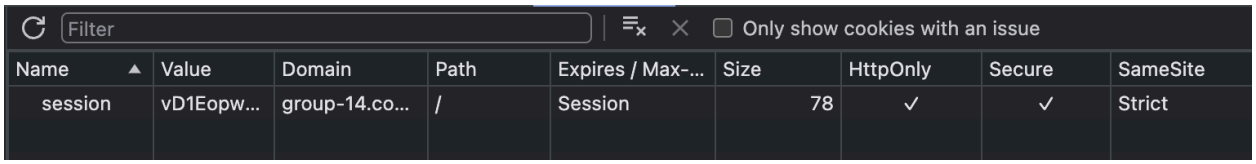
```
def contains_sqli_attempt(input_string):
```

```
    SQLI_BLACKLIST = ["'", '"', ";", "--", "/*", "*/", "=", "%"]
    return any(char in input_string for char in SQLI_BLACKLIST)
```

- **Session Cookies Extension**

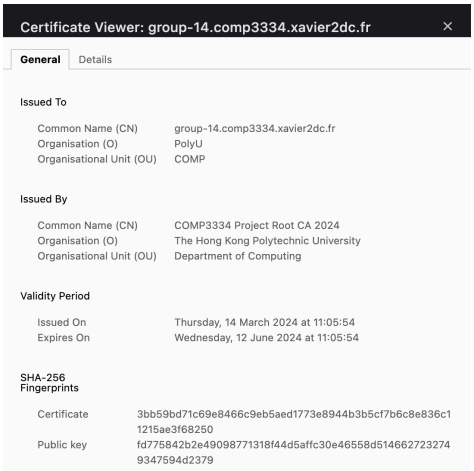
  Another aspect of the security framework in E2EE applications is that they should not be vulnerable to cross-site request forgery, under this case, we add the sameSite to be strict into the session cookie.

  Also, to prevent the XSS attack we add the Httponly for yes, and secure into the cookie, and this will be helpful.



5.1.5 HTTPS Protection

The End-to-end Encrypted Chat Web Application also uses the HTTPS to make the website communication, which is encrypted by TLS version 1.3, and use the trusted Certificate Authority (CA) which is provided by the professor to generate the website certification with the common domain name: group-14.comp3334.xavier2dc.fr. The certification uses the algorithms - X9.62 ECDSA to generate the signature and use SHA-384 to calculate the hashing value. After generating the certification, use the Nginx Web Server to host the website and use port 8443 to be the HTTPS port, if users access the website with this port and include the HTTPS link, will be able to communicate safely online.



6. CONCLUSIONS

This was a useful end-to-end encrypted communication tool security project, where we learned many practical skills for developing secure and effective systems and how end-to-end encrypted chatting came about. Debugging can be painful sometimes, but exciting when we find a solution. The results show that our platform successfully combines strong security practices and makes the transmission of information more secure.

We are grateful for this project and everything we learned from it, especially some of the mechanics of security and application security.

7. REFERENCES

Guido Van Rossum, & Drake, F. L. (2011). *The Python language reference manual : for Python version 3.2*. Network Theory Ltd.

HMAC (hash-based message authentication codes) definition. Okta. (2023, September 15). https://www.okta.com/identity-101/hmac/

Katz, J. (2019). *Introduction To Modern Cryptography.* CRC Press.

Patil, V. (2021, December 27). *How time-based one-time password (TOTP) algorithm works*. LinkedIn. https://www.linkedin.com/pulse/how-time-based-one-time-password-totp-algorithm-works-vijay-patil