COMP3334

Computer Systems Security

Group Project

Group 42

Group members:

23025024D Kong Pak Kwan

23026258D Chang Chi Yan

23021927D Hui Shing Hin

23039798D Yau Chi Kin

Table of Contents

Table of Contents

Authentication 3

End-to-end Encrypted (E2EE) 9

TLS 22

Authentication

The chat system has been implemented with user authentication according to NIST guidelines. Libraries imported for Authentication in Python Flask:

- pyotp - For generating One Time Password
- qrcode - For generating QR Code for One Time Password
- String - For formatting recovery key
- Hashlib - For hashing passwords
- os - For generating cryptological secure characters
- binascii - For formatting salt for easier database storage
- requests - For requesting check during reCAPTCHA and Have I been pwned
- flask_limiter and flask_limiter.util - For rate limiting mechanisms

1. Registration system and user-chosen memorized secret (password)

    1.1 For username, the system will do a validation check on whether it is unique.

    1.2 For passwords, the system will do a validation check on:

    - *Character length:* it must be at least 8 characters long
    - *Password verifiers:* password is checked against "Have I Been Pwned" database and rejects all compromised passwords.

```python
    cur.execute("SELECT username FROM users WHERE username=%s", (username,))
    checkUserName = cur.fetchone()

    #check unique username
    if(checkUserName):
        error = 'Username is taken'
        return render_template('register.html', error=error)
```

Fig 1: Unique username check with database

```python
    #password must be at least 8 char
    if len(password) < 8:
        error = 'Password must be at least 8 characters.'
        return render_template('register.html', error=error)
```

Fig 2: Password length check

```python
#Have I Been Pwned
def passwordSecurityCheck(password):
    #the website uses fiirst 5 characters to search the database
    #if exists, then it will check the count of suffix appearing in that line
    sha1Hash = hashlib.sha1(password.encode("utf-8")).hexdigest().upper()
    prefix = sha1Hash[:5]
    suffix = sha1Hash[5:]
    url = f"https://api.pwnedpasswords.com/range/{prefix}"
    response = requests.get(url)
    if response.status_code == 200:
        for line in response.text.splitlines():
            linePrefix, count = line.split(":")
            if linePrefix == suffix:
                return int(count)
    return 0
```

Fig 3: Check with "Have I been Pwned" compromised password database

2. Database Security on Passwords

When a new account is created, the password is salted and hashed before inserting into the database.

2.1 Hash: pbkdf2_hmac with sha256 is used for the hash. pbkdf2 is computationally slow but are far more computationally secure compared to easier encryptions such as SHA.

2.2 Salt: using os.random(32), a secure salt is generated and they are put into hashing process

Fig 4: salt and hash process

the salt and hashed password will then be stored inside the database. Salt is unique per user as it is generated every time a user registers.

2.3 For login system, the password input field is then salted and hashed. Then it will check against the hashed password inside the database.

```
#salt and hash password
def generateHashedPassword(password):
        salt = os.urandom(32)
        #reformat to be stored in database
        saltHex = binascii.hexlify(salt).decode()
        hashedPassword = hashlib.pbkdf2_hmac('sha256', password.encode('utf-8'), salt, 100000)
        return (saltHex, hashedPassword)
```

Fig 4: Salt and hash process during registration

```
            cur.execute("SELECT user_id, password, salt FROM users WHERE username=%s", (username,))
            account = cur.fetchone()
            if(account):
                databasePassword = account[1]
                saltHex = account[2]
                #reformat to the original salt
                salt = binascii.unhexlify(saltHex)
                #hash and salt the entered password so can check with database
                hashedPassword = hashlib.pbkdf2_hmac('sha256', password.encode('utf-8'), salt, 100000)
                if (str(hashedPassword) == str(databasePassword)):
                    return redirect(url_for('otp', username=username))
                else:
                    error = 'Invalid credentials'
```

Fig 5: Salt and hash, then compare with input during login process

3. One-time-password (OTP), recovery key, sessions

   When an account is created, user will be directed to a page showing One time password key, its QR code and the recovery key. User can use a authenticator to scan the QR code or copy the key to the app. And mark down the recovery key in case the user loses his credentials.

```python
#Shows QR Code, secret key, recovery key for first time users (otpFirstTIme.html)
@app.route('/register/otpFirstTime', methods=['GET', 'POST'])
def otpFirstTime():
    username = request.args.get('username', None)
    secretKey = request.args.get('secretKey', None)
    recoveryKey = request.args.get('recoveryKey', None)
    uri = pyotp.TOTP(secretKey).provisioning_uri(name=username, issuer_name="COMP3334 Group Project")
    qrcode.make(uri).save("static/images/qrcode.png")
    if request.method == 'POST':
        flash('You have registered successfully.', 'info')
        return redirect(url_for('login'))
    return render_template('otpFirstTime.html', username=username, secretKey=secretKey, recoveryKey=recoveryKey)
```

Fig 6: Code of the keys

Fig 7: otpFirstTime Page

After entering credentials, the website will ask for the OTP key from the user. If the user has lost the OTP, he can click "Lost OTP?" and it is redirected to the recovery key page. Entering the correct recovery key will show the otpFirstTimePage again, secret OTP key and recovery key is refreshed and therefore is new.

Sessions are only started once the OTP is correct. so users will not connect session after they have entered correct credientals yet.

```python
#Ask for OTP, verifies OTP (otp.html)
@app.route('/otp', methods=['GET', 'POST'])
def otp():
        error = None
        username = request.args.get('username', None)
        if request.method == 'POST':
            enteredOtp = str(request.form.get('otp'))
            #get the user_id and secret key stored in database
            cur = mysql.connection.cursor()
            cur.execute("SELECT user_id, otp_key FROM users WHERE username=%s", (username,))
            account = cur.fetchone()
            if account:
                otp_key = account[1]
                databaseOtp = pyotp.TOTP(otp_key)
                #compare otp, if correct, establish session
                if (databaseOtp.now() == enteredOtp):
                    session['username'] = username
                    session['user_id'] = account[0]
                    return redirect(url_for('index'))
            error = 'Invalid One-time Password'
        return render_template('otp.html', username=username, error=error)
```

Fig 8: OTP verification page

```python
#verify recovery key
@app.route('/recoverycheck', methods=['GET', 'POST'])
def recoveryCheck():
    error = None
    username = request.args.get('username', None)
    if request.method == 'POST':
        enteredRecovery = request.form.get('recoveryKey')
        #get recovery key from database
        cur = mysql.connection.cursor()
        cur.execute("SELECT recovery_key FROM users WHERE username=%s", (username,))
        recoveryKey = cur.fetchone()
        databaseRecovery = recoveryKey[0]
        #check database and entered recovery key
        if databaseRecovery == enteredRecovery:
            #regenerate a new set of secretKey, recoverykey
            secretKey = pyotp.random_base32()
            recoveryKey = generateRecoveryKey()
            cur = mysql.connection.cursor()
            cur.execute("UPDATE users Set otp_key = %s,  recovery_key = %s WHERE username=username", (secretKey, recovery
            mysql.connection.commit()
            return redirect(url_for('otpFirstTime', username=username, secretKey=secretKey, recoveryKey=recoveryKey))
        else:
            error = 'Incorrect recovery key'
    return render_template('recovery.html', username=username, error=error)
```

Fig 9: recovery key page

4. reCAPTCHA

reCAPTCHA is set during the login screen. It uses images to authenticate.

```
<!-- reCAPTCHA check box -->
<div class="g-recaptcha" data-sitekey="6Le95aIpAAAAAKGi1ng068FWdwuxylMh3iH_CIpR"></div>
```

Fig 10: the html to print the reCAPTCHA prompt

Fig 11: The Google reCAPTCHA page and its key, allowed domain names

5. Rate Limit Mechanisms

Flask_limiter is used to limit the usage of users. By default, we have limited 50 access per hour and 200 per day.

In chat.html, the limiter is disabled such that the user will not be blocked after sending 50 messages.

```
#limiter: default 200 per day, 50 per hour
limiter = Limiter(
    get_remote_address,
    app=app,
    default_limits=["200 per day", "50 per hour"],
    storage_uri="memory://",
)
```

Fig 12: General Code for Flask Limiter

End-to-end Encrypted (E2EE)

The chat system has been implemented with user authentication according to NIST guidelines. The functions for generating ECDH and deriving Shared Secret are stored in KeyGeneration.js, other code is written in either otpFirstTime.html or chat.html using JavaScript. Libraries imported for Authentication in JavaScript:

- crypto - For WebCryptoAPI

1. Use the ECDH key exchange protocol to establish a shared secret between two users, then generate the encryption key and mac key by the shared secert

   When a new account is created, a private key will generate and save in the local storage, the public key will generate and send to the database.

```
// ECDH
async function generateKeyPair() {
  const keyPair = await crypto.subtle.generateKey(
    {
      name: 'ECDH',
      namedCurve: 'P-384'
    },
    true,
    ['deriveKey', 'deriveBits']
  );

  return keyPair;
}
```

Fig 13: Generate ECDHs, P-384 (384-bit) is used.

```
async function keyGeneration(username) {
  const KeyPair = await generateKeyPair();
  const exportedPublicKey = await window.crypto.subtle.exportKey('spki', KeyPair.publicKey);
  const StringPublicKey = arrayBufferToBase64(exportedPublicKey);

  // Set the ECDH public key in the hidden input field
  document.getElementById("ecdhPublicKey").value = StringPublicKey;
  // Export the ECDH private key
  const exportedPrivateKey = await window.crypto.subtle.exportKey('pkcs8', KeyPair.privateKey);
  const StringPrivateKey = arrayBufferToBase64(exportedPrivateKey);
  localStorage.setItem(`ecdhPrivateKey_${username}`, StringPrivateKey);
  console.log("Private key:", KeyPair.privateKey);
  console.log("PublicKey key:", KeyPair.publicKey);
  console.log("PublicKey key in Base64:"+ document.getElementById("ecdhPublicKey").value);
  console.log("Private key in Base64:"+ localStorage.getItem(`ecdhPrivateKey_${username}`));
}
```

Fig 14: Code for storing ECDH keys.

```python
def sendECDH():
    ecdhPublicKey = request.form.get('ecdhPublicKey')  # Retrieve the ecdhPublicKey value from the form
    username = request.form.get('name')
    cur = mysql.connection.cursor()
    cur.execute("UPDATE users SET ECDH_publicKey = %s WHERE username = %s", (ecdhPublicKey, username))
    mysql.connection.commit()


#Shows QR Code, secret key, recovery key for first time users (otpFirstTIme.html), generate and sending the key
@app.route('/register/otpFirstTime', methods=['GET', 'POST'])
def otpFirstTime():
    username = request.args.get('username', None)
    secretKey = request.args.get('secretKey', None)
    recoveryKey = request.args.get('recoveryKey', None)
    uri = pyotp.TOTP(secretKey).provisioning_uri(name=username, issuer_name="COMP3334 Group Project")
    qrcode.make(uri).save("static/images/qrcode.png")
    if request.method == 'POST':
        sendECDH()
        flash('Public key updated successfully.', 'info')
        return redirect(url_for('login'))
    return render_template('otpFirstTime.html', username=username, secretKey=secretKey, recoveryKey=recoveryKey)
```

Fig 15: Code for Sending public keys when the register is done.

```javascript
// Initial execution when the document is ready
$(document).ready(function() {
    fetch('/users')
        .then(response => response.json())
        .then(data => {
            data.users.forEach(user => {
                let userId = user[0];
                let username = user[1];
                userInfo[userId] = username;
                console.log('user information:',user);
                localStorage.setItem(`publickey_${username}`, user[2]);
            });
            populateUsers(data.users);
        })
        .catch(error => console.error('Error fetching user info:', error));
});
```

Fig 16: Code that getting user information, get the public key from database (store at local storage)

Once the user selects who wants to communicate, shared secret, encryption key, mac key will be generated and stored as variables for later use. Also, store the salt and iv as variables.

```
// Handle user selection change
document.getElementById('userList').addEventListener('change', event => {
    peer_id = parseInt(event.target.value, 10); // Convert selected peer_id to integer
    peer_name = userInfo[peer_id];
    clearChatBox();
    lastMessageId = 0;
    last_iv = 0;
    salt = 0;
    if (localStorage.getItem(`ecdhPrivateKey_${myusername}`)) {
        Encryption_MAC_Key(localStorage.getItem(`ecdhPrivateKey_${myusername}`),localStorage.getItem(`publickey_${peer_name}`));
    } else {
        alert(`Local storage is lost.
        Generating a new private key and public key is crucial for secure communication.
        Without generating new keys, your communication may be vulnerable to unauthorized access.`);
    }
    fetchMessages(); // Fetch messages for the new selection
});
```

Fig 17: default values and set the useful variables for later use

```
function Encryption_MAC_Key(StringPrivateKey, StringPublicKey) {
    let publicKey = base64ToArrayBuffer(StringPublicKey);
    let privateKey = base64ToArrayBuffer(StringPrivateKey);

    console.log('public key for derive key:',publicKey);
    console.log('private key for derive key:',privateKey);

    let aesgcmInfo = textEncoder.encode(`CHAT_KEY_${myusername}_to_${peer_name}`);
    let macInfo = textEncoder.encode(`CHAT_MAC_${myusername}_to_${peer_name}`);

    deriveSharedSecret(privateKey, publicKey)
  .then((sharedSecret) => {
    using_sharedSecret = sharedSecret;
    console.log(`Shared secret for user ${peer_name}:`, new Uint8Array(sharedSecret));
    return deriveKeysHKDF(sharedSecret, numberToUint8Array(salt));
  })
  .then((keys) => {
    using_encryptionKey = keys.encryptionKey;
    using_macKey = keys.macKey;
    console.log('Encryption Key:', using_encryptionKey);
    console.log('MAC Key:', using_macKey);
  })
  .catch((error) => console.error('Error when generate shared secret and keys:', error));
}
```

Fig 18: Code for generate encryption key and mac key

```
async function deriveSharedSecret(privateKey, publicKey) {
  try {
    const importedPrivateKey = await crypto.subtle.importKey(
      'pkcs8',
      privateKey,
      { name: 'ECDH', namedCurve: 'P-384' },
      true,
      ['deriveBits']
    );

    const importedPublicKey = await crypto.subtle.importKey(
      'spki',
      publicKey,
      { name: 'ECDH', namedCurve: 'P-384' },
      true,
      []
    );

    const sharedSecret = await crypto.subtle.deriveBits(
      {
        name: 'ECDH',
        public: importedPublicKey,
      },
      importedPrivateKey,
      256
    );

    return sharedSecret;
  } catch (error) {
    throw new Error('Error deriving shared secret: ' + error.message);
  }
}
```

Fig 19: Code for generate Shared Secret (256 bits)

```javascript
async function deriveKeysHKDF(sharedSecret, salt) {
  try {
  const sharedSecretKey = await crypto.subtle.importKey(
    'raw',
    sharedSecret,
    { name: "HKDF" },
    false,
    ['deriveKey']
  );

  const encryptionKey = await crypto.subtle.deriveKey(
    {
      name: 'HKDF',
      salt: salt,
      info: aesgcmInfo,
      hash: { name: 'SHA-256' },
    },
    sharedSecretKey,
    { name: 'AES-GCM', length: 256 },
    true,
    ['encrypt', 'decrypt']
  );

  const macKey = await crypto.subtle.deriveKey(
    {
      name: 'HKDF',
      salt: salt,
      info: macInfo,
      hash: { name: 'SHA-256' },
    },
    sharedSecretKey,
    { name: 'HMAC', hash: { name: 'SHA-256' }, length: 256 },
    false,
    ['sign', 'verify']
  );

  return { encryptionKey, macKey };
} catch (error) {
  throw new Error('Error deriving keys using HKDF: ' + error.message);
```

Fig 20: Code for generate encryption key (256 bits) and mac key (256 bits)

2. When the user clicks “Send” in the chat page, the message will be encrypted by the encryption key, and sign the iv for protecting the iv. Then store the encrypted message, iv, signed iv in one Json and store this with peer_id, myID, and key_refresh in payload. Finally, send this Json to database.

```
// Send message function
function sendMessage() {
    if (peer_id == -1) return; // Exit if no peer selected
  const message = document.getElementById('messageInput').value;
  const iv = numberToUint8Array(last_iv + 1);

  encryptMessage(message, using_encryptionKey, iv, using_macKey)
    .then(encryptedMessage => {
      const payload = {
        receiver_id: peer_id,
        sender_id: myID,
        message_text: encryptedMessage,
        key_refresh: "false",
      };

      return sendMessageToServer(payload);
    })
    .then(() => {
      document.getElementById('messageInput').value = '';
    })
    .catch(error => {
      // Handle any errors from sending the message
      console.error('Error sending message:', error);
    });
}
```

Fig 21: Code to prepare message

```
async function encryptMessage(message, using_encryptionKey, iv, using_macKey) {
    console.log('Message:', message);
    console.log('Using Encryption Key:', using_encryptionKey);
    console.log('IV:', iv);
    console.log('Using MAC Key:', using_macKey);
    const encoder = new TextEncoder();
    const data = encoder.encode(message);
    const additionalData = new TextEncoder().encode(`Chat_MSG_${myID}_to_${peer_id}`);
    console.log('message sent:'+message);

    const encryptedData = await crypto.subtle.encrypt(
        { name: 'AES-GCM', iv, additionalData: additionalData, tagLength: 128 },
        using_encryptionKey,
        data
    );

    // Generate HMAC-SHA256 for the IV using the MAC key
    const ivHmac = await crypto.subtle.sign({ name: 'HMAC', hash: { name: 'SHA-256' } }, using_macKey, iv);

    const serializableEncryptedMessage = {
    encryptedData: Array.from(new Uint8Array(encryptedData)).join(','),
    ivHmac: Array.from(new Uint8Array(ivHmac)).join(','),
    iv: Array.from(new Uint8Array(iv)).join(','),
    };

return serializableEncryptedMessage;
}
```

Fig 22: Code for encrypting messages

```
//the part that send message to server
function sendMessageToServer(payload) {
  return fetch('/send_message', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(payload),
  })
    .then(response => {
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      return response.json();
    })
    .then(data => {
      console.log('Message sent:', data);
      return data; // Return the parsed JSON response
    })
    .catch(error => {
      console.error('Error sending message to server:', error);
      throw error; // Re-throw the error for error handling
    });
}
```

Fig 23: Code for sending messages to database

```python
@app.route('/send_message', methods=['POST'])
def send_message():
    if not request.json or not 'message_text' in request.json:
        abort(400)  # Bad request if the request doesn't contain JSON or lacks 'message_text'
    if 'user_id' not in session:
        abort(403)

    # Extract data from the request
    sender_id = session['user_id']
    receiver_id = request.json['receiver_id']
    message_text = request.json['message_text']
    key_refresh = request.json['key_refresh']

    # Assuming you have a function to save messages
    save_message(sender_id, receiver_id, message_text, key_refresh)

    return jsonify({'status': 'success', 'message': 'Message sent'}), 200

def save_message(sender, receiver, message, key_refresh):
    cur = mysql.connection.cursor()
    cur.execute("INSERT INTO messages (sender_id, receiver_id, message_text, key_refresh) VALUES (%s, %s, %s, %s)",
                (sender, receiver, message, key_refresh))
    mysql.connection.commit()
    cur.close()
```

Fig 24: Python for sending messages to database

3. Decrypt and display the message one by one

The signed IV, message, iv will get from the message. Check the iv is greater than the last one, then verify the signed iv, after that decrypt the message

```python
@app.route('/fetch_messages')
@limiter.exempt
def fetch_messages():
    if 'user_id' not in session:
        abort(403)

    last_message_id = request.args.get('last_message_id', 0, type=int)
    peer_id = request.args.get('peer_id', type=int)

    cur = mysql.connection.cursor()
    query = """SELECT message_id,sender_id,receiver_id,message_text,key_refresh FROM messages
               WHERE message_id > %s AND
               ((sender_id = %s AND receiver_id = %s) OR (sender_id = %s AND receiver_id = %s))
               ORDER BY message_id ASC"""
    cur.execute(query, (last_message_id, peer_id, session['user_id'], session['user_id'], peer_id))

    # Fetch the column names
    column_names = [desc[0] for desc in cur.description]
    # Fetch all rows, and create a list of dictionaries, each representing a message
    messages = [dict(zip(column_names, row)) for row in cur.fetchall()]

    cur.close()
    return jsonify({'messages': messages})
```

Fig 25: Get the message from the message table

```javascript
function fetchMessages() {
  if (peer_id === -1 || processingMessages) return; // Exit if no peer selected or messages are being processed
  processingMessages = true; // Set flag to indicate that messages are being processed

  fetch(`/fetch_messages?last_message_id=${lastMessageId}&peer_id=${peer_id}`)
    .then(response => response.json())
    .then(data => {
      const messages = data.messages.sort((a, b) => a.message_id - b.message_id);
      let index = 0;

      const displayNextMessage = () => {
        if (index < messages.length) {
          const message = messages[index];
          displayMessage(message);
          lastMessageId = message.message_id;
          index++;
          processNextMessage();
        } else {
          processingMessages = false; // Set flag to indicate that message processing is complete
        }
      };

      const processNextMessage = () => {
        setTimeout(displayNextMessage, 10);
      };

      processNextMessage();
    })
    .catch(error => {
      console.error('Error fetching messages:', error);
      processingMessages = false; // Reset flag in case of error
    });
}
```

Fig 26: get every message related to sender and receiver

```
if (ivArray.length === 0) {
    // Handle the case where the received IV is empty
    throw new Error("Received IV is empty");
} else {
    received_iv = ivArray.reduce((a, b) => a * 10 + b, 0);

    if (received_iv > last_iv) {
        console.log("Received IV is greater than the last IV");
    } else {
        throw new Error("Received IV is not greater than the last IV");
    }
}

const encryptedData = new Uint8Array(encryptedDataArray).buffer;
const ivHmac = new Uint8Array(ivHmacArray).buffer;
const iv = new Uint8Array(ivArray).buffer;


// Decrypt the message using AES-GCM
decryptMessage(encryptedData, using_encryptionKey, ivHmac, iv, using_macKey, message.sender_id, message.receive
    .then(decryptedMessage => {
        if (decryptedMessage == '/Decryption failed'){
            if (decryption_fail == false){
                messageElement.textContent = `Failed to decrypt the messages`;
                messagesContainer.appendChild(messageElement);
                decryption_fail = true;
            }
        } else {
            decryption_fail = false;
            last_iv = received_iv;
            messageElement.textContent = `From ${sender} to ${receiver}: ${decryptedMessage}`;
            messagesContainer.appendChild(messageElement);
        }
    })
     .catch(error => console.error('Error decrypting message:', error));
```

Fig27: decrypt process in the display function and check the iv

```
<script type="text/javascript">
async function decryptMessage(encryptedMessage, encryptionKey, ivHmac, iv, macKey, sender, receiver) {
    const additionalData = new TextEncoder().encode(`Chat_MSG_${sender}_to_${receiver}`);

    // Verify the integrity of the IV using the provided HMAC
    const IvHmacVerified = await crypto.subtle.verify(
        { name: 'HMAC', hash: { name: 'SHA-256' } },
        macKey,
        ivHmac,
        iv
    );

    if (!IvHmacVerified) {
        console.error('IV verification failed');
        return '/Decryption failed';
    }

    try {
        const decryptedMessage = await crypto.subtle.decrypt(
            {
                name: 'AES-GCM',
                iv: iv,
                additionalData: additionalData,
                tagLength: 128
            },
            encryptionKey,
            encryptedMessage
        );

        // Convert the decrypted message to a readable string
        const messageText = new TextDecoder().decode(decryptedMessage);

        // Return the decrypted message
        return messageText;
    } catch (error) {
        console.error('Error decrypting message:', error);
        // Handle the decryption failure here, such as returning an error message
```

Fig28: verified and decrypt the message

4. When the user clicks the refresh button, send a special message in the display message process, when a request refresh message is detected, it will update the salt and iv, then generate new encryption key and mac key for the following decryption.

```
Refresh Keys function placeholder
nc function refreshKeys() {
 // sign with old macKey
 const OldHmac = await crypto.subtle.sign({ name: 'HMAC', hash: { name: 'SHA-256' } }, using_macKey, StringChange);
 deriveKeysHKDF(using_sharedSecret, numberToUint8Array(salt+1))
 .then(async (keys) => {
     // sign with new macKey
     const NewHmac = await crypto.subtle.sign({ name: 'HMAC', hash: { name: 'SHA-256' } },  keys.macKey, StringChange);

     const refresh_message = {
         NewHmac: Array.from(new Uint8Array(NewHmac)).join(','),
         OldHmac: Array.from(new Uint8Array(OldHmac)).join(','),
     };
     const payload = {
         receiver_id: peer_id,
         sender_id: myID,
         message_text: refresh_message,
         key_refresh: "true",
     };

     sendMessageToServer(payload)
     .catch(error => {
         // Handle any errors from sending the message
         console.error('Error sending refresh message:', error);
     });
 })
 .catch((error) => console.error('Error when generate keys:', error));
```

Fig29: the code for send refresh key message

```
    const OldHmac = await crypto.subtle.verify(
        { name: 'HMAC', hash: { name: 'SHA-256' } },
        using_macKey,
        OldHmacArray,
        StringChange
    );
    deriveKeysHKDF(using_sharedSecret, numberToUint8Array(salt+1))
    .then(async (key) => {
        //sign with new macKey
        const NewHmac = await crypto.subtle.verify(
            { name: 'HMAC', hash: { name: 'SHA-256' } },
            key.macKey,
            NewHmacArray,
            StringChange
        );
        if (!OldHmac || !NewHmac) {
            console.error('Verification failed');
        } else {
            using_encryptionKey = key.encryptionKey;
            using_macKey = key.macKey;
            console.log('New Encryption Key:', using_encryptionKey);
            console.log('New MAC Key:', using_macKey);
            last_iv = 0;
            salt = salt +1;
            messageElement.textContent = "The key has been refreshed";
            messagesContainer.appendChild(messageElement);
        }
    })
    .catch((error) => console.error('Error when refreshing the key:', error));
    return;
}
```

Fig30: the code for checking the message and refreshing key

TLS

To generate the key called group42.pem, we typed the following:

- openssl ecparam -name secp384r1 -genkey -out group42.pem

In group42.conf, we set up specifications for the SSL certificate later.

```
[req]
default_bits        = 384
default_md          = sha384
prompt              = no
encrypt_key         = no
distinguished_name  = dn
req_extensions      = req_ext
x509_extensions     = v3_ca

[dn]
CN                  = group-42.comp3334.xavier2dc.fr
C                   = HK

[req_ext]
subjectAltName      = @alt_names

[v3_ca]
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid:always,issuer
basicConstraints        = critical,CA:FALSE
keyUsage                = critical,digitalSignature
extendedKeyUsage        = serverAuth
subjectAltName          = @alt_names

[alt_names]
DNS.1 = group-42.comp3334.xavier2dc.fr
```

Fig 31: SSL Configuration File

Command to generate our certificate with the SSL configuration file:

```
D:\STUDY\comp3334\bin>openssl x509 -req -days 90 -in group42.csr -CA cacert.crt -CAkey cakey.pem -CAcreateserial -out group42.crt -extensions v3_ca -extfile group42.conf -sha384
Certificate request self-signature ok
subject=CN=group-42.comp3334.xavier2dc.fr
```

The content of group-42 certificate:

```
D:\STUDY\comp3334\bin>openssl x509 -in group42.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            5c:ad:fa:89:92:83:5b:cf:ca:fd:6d:c8:1a:06:24:1b:4f:4c:40:16
        Signature Algorithm: ecdsa-with-SHA384
        Issuer: C=HK, O=The Hong Kong Polytechnic University, OU=Department of Computing, CN=COMP3334 Project Root CA 2024
        Validity
            Not Before: Apr  7 12:20:07 2024 GMT
            Not After : Jul  6 12:20:07 2024 GMT
        Subject: CN=group-42.comp3334.xavier2dc.fr
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:4e:ed:f7:42:09:03:bb:58:b3:d1:1f:82:0e:e7:
                    8e:41:8c:11:6f:ca:39:2f:b6:56:d9:3d:78:65:3b:
                    02:52:15:be:87:d5:66:c3:25:68:f5:5b:2e:0b:d0:
                    5e:5d:81:ba:db:74:ac:59:66:b9:c2:73:57:12:d4:
                    35:95:98:95:2c:0e:5b:c4:af:11:e4:b8:cb:f3:ba:
                    ba:98:97:b0:37:5f:5f:81:c5:ce:94:9b:82:41:6d:
                    ad:a6:13:68:02:f3:a2
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                BC:CD:02:F2:23:B7:E0:6A:CB:BC:40:53:2B:F4:55:27:91:DE:9D:28
            X509v3 Authority Key Identifier:
                3B:4E:1B:40:FD:B5:1C:FF:7C:33:DB:B6:FB:AF:3C:BC:EC:24:2B:CE
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Key Usage: critical
                Digital Signature
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Alternative Name:
                DNS:group-42.comp3334.xavier2dc.fr
    Signature Algorithm: ecdsa-with-SHA384
    Signature Value:
        30:81:87:02:41:47:bb:86:c1:e8:13:31:81:65:be:bc:a6:42:
        9c:d6:7d:3b:6a:b2:7f:f9:29:0b:24:97:8e:1b:6a:73:00:ef:
        a0:a0:e2:47:bc:c9:3d:49:2b:71:05:f3:36:a4:95:fd:1e:bd:
        05:b8:68:b3:36:71:65:5c:85:8c:0e:4a:d7:0d:04:9f:02:42:
        00:96:db:c7:f1:cd:8c:e6:08:aa:f0:54:51:ff:88:cb:96:74:
        e4:67:ac:b9:9b:f3:e8:76:c6:a9:98:3a:7b:a3:0a:69:c4:f0:
        96:41:7a:09:c4:36:22:dd:b8:33:d0:1c:72:d6:51:8e:c5:26:
        f6:2e:f1:e6:6d:c2:06:65:cf:32:58:3e
```

Nginx.conf: configuration of the nginx server

```
events {}

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

    server {
        listen 8443 ssl;
        server_name group-42.comp3334.xavier2dc.fr;

        # SSL configuration
        ssl_certificate /etc/nginx/certs/group42.crt;
        ssl_certificate_key /etc/nginx/certs/group42.pem;
        ssl_session_timeout 5m;

        # HSTS (ngx_http_headers_module is required) (604800 seconds(1 week)) task 3.5
        add_header Strict-Transport-Security "max-age=604800" always;


        # only TLSv1.3 used task3.1
        ssl_protocols TLSv1.3;
        ssl_prefer_server_ciphers off;
        # task 3.3
        ssl_conf_command Options PrioritizeChaCha;
        ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256;

        # task 3.2
        ssl_ecdh_curve X25519;

        # task 3.4
        ssl_stapling off;
        ssl_stapling_verify off;


        resolver 127.0.0.1;

        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;
        }
    }
}
```

After applying the TLS:

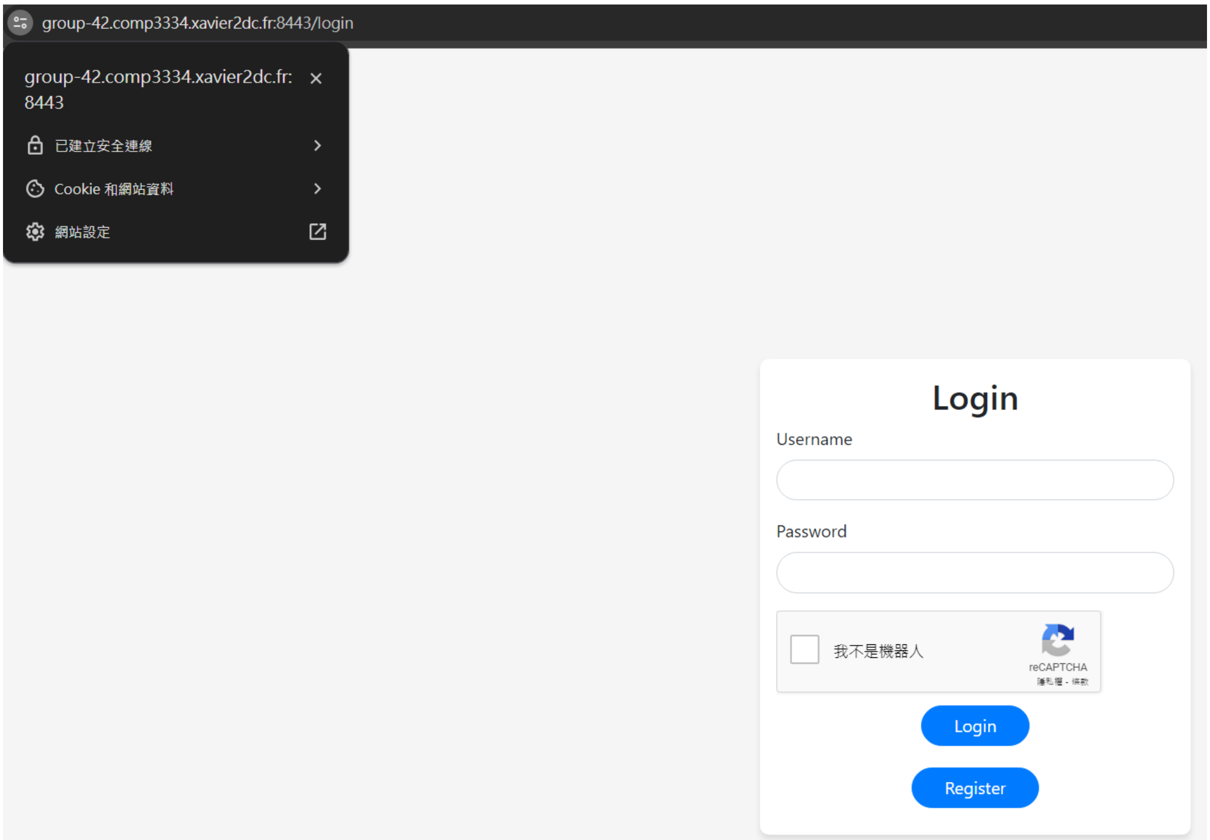

Fig 32: The new HTTP

group-42.comp3334.xavier2dc.fr

Subject Name

Common Name	group-42.comp3334.xavier2dc.fr

Issuer Name

Country	HK
Organization	The Hong Kong Polytechnic University
Organizational Unit	Department of Computing
Common Name	COMP3334 Project Root CA 2024

Validity

Not Before	Thu, 04 Apr 2024 05:45:36 GMT
Not After	Wed, 03 Jul 2024 05:45:36 GMT

Subject Alt Names

DNS Name	group-42.comp3334.xavier2dc.fr

Public Key Info

Algorithm	Elliptic Curve
Key Size	384
Public Value	04:4E:ED:F7:42:09:03:BB:58:B3:D1:1F:82:0E:E7:8E:41:8C:11:6F:CA:39:2F:B6:...

Miscellaneous

Serial Number	5C:AD:FA:89:92:83:5B:CF:CA:FD:6D:C8:1A:06:24:1B:4F:4C:40:14
Signature Algorithm	ECDSA with SHA-384
Version	3
Download	PEM (cert) PEM (chain)


憑證
一般
詳細資料
憑證路徑
憑證資訊
這個憑證的使用目的如下:
• 確保遠端電腦的識別
發給: group-42.comp3334.xavier2dc.fr
簽發者: COMP3334 Project Root CA 2024
有效期自 7/4/2024 到 6/7/2024
安裝憑證(I)...
簽發者聲明(S)
確定