COMP 3334 Group Project

End-to-end encrypted chat web application

Group 31

CHEN Ziyang 21095751d

FANG Yuji 21105368d

HE Rong 21101622d

LI Shuhang 21102658d

Qin Qipeng 21101226d

Table of Contents:

1. Introduction

Our project aims to create a web application with end-to-end encrypted communication. And this web application complies with several security guidelines provided in NIST Special Publication 800-63B. Servers that enable the connection are also unable to decrypt the content.

Our project's key characteristics may be broken down into three primary points. Secure Multi-Factor Authentication (MFA) comes first. To provide an extra degree of protection against unwanted access, we use a two-step verification method that combines a recovered key or one-time password (OTP) mechanism with an internal secret (password). Additionally, even if password data is hacked, employing Bcrypt assures.

End-to-End (E2EE) encryption comes in second. We guarantee that communications are encrypted on the sender's device and only decrypted on the recipient's device by implementing E2EE, which prevents intermediary servers from deciphering the messages.

Additionally, the current TLS configuration. The most recent TLS protocols are used to encrypt all communications between the client and the server to guard against manipulation and eavesdropping. Describe how user passwords are kept safe and validated, which libraries we use, how key materials are derived and stored, and how domain certificates are created.

We will examine the specifics of these features' implementation, the difficulties encountered, and the solutions used to satisfy the project's security needs throughout this report.

2. List of NIST SP 800-63B Requirements Implement

2.1 Secure MFA mechanism based on passwords and OTP

In our system, the user needs to enter username and password, which is the basic authentication step and belongs to the first factor authentication. There is also an input field called Second Password, which is bound to a drop-down menu where the user can select Google Authenticator or Recovery Key as the second authentication method. This is the second factor in the MFA. Here we are using Google Authenticator to generate time-based OTP, which belongs to the Single-Factor OTP Device.

This program creates an OTP in response to a timer or counter to provide an extra degree of protection to the login procedure. In case the user chooses Google Authenticator, they will have to input a one-time password that is created by the application.

2.1.1 User-chosen Memorized Secret

The memorized password chosen by the user, often referred to as a password, is the basic security measure in the authentication system. In our implementation, passwords are used as the primary method of user authentication.

To handle passwords, we use the register and check_login functions (app.py). During the registration process, the password is received from the user, security standards are checked (for example, whether the password has been previously leaked), and then securely stored in the database.

```
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.json['username']
        #password = request.json['password']
        password = request.json['password'].encode('utf-8')  # Update: Encode the password to bytes
        recoverykey = request.json['recoverykey']
        # Update: Generate a salt and hash the password
        hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
        cur = mysql.connection.cursor()
        cur.execute("SELECT user_id FROM users WHERE username=%s", (username,))
        account = cur.fetchone()
        if not account:
            if len(password) < 8:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password should be longer than 8 letters'}), 200
            if check_password(password):
                b32_password = base64.b32encode(password.encode()).decode()
                qrcode = pyotp.totp.TOTP(b32_password).provisioning_uri(name=username, issuer_name='WebApp')
                cur = mysql.connection.cursor()
                # cur.execute("INSERT INTO users (username, password,recoverykey,ratelimit) VALUES (%s, %s, %s, %s)", (username, pass
                cur.execute("INSERT INTO users (username, password, secretkey, recoverykey) VALUES (%s, %s, %s, %s)",
                            (username, hashed_password, pyotp.random_base32(), recoverykey))
                mysql.connection.commit()
                return jsonify({'status': 'success', 'message': 'Register Successed', "data": qrcode}), 200
            else:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password has been leaked'}), 200
        else:
            return jsonify({'status': 'error', 'message': 'User has registered'}), 200
    return jsonify({'status': 'success', 'message': 'Register Successed'}), 200
```

At login, the password is verified in the check_login function.

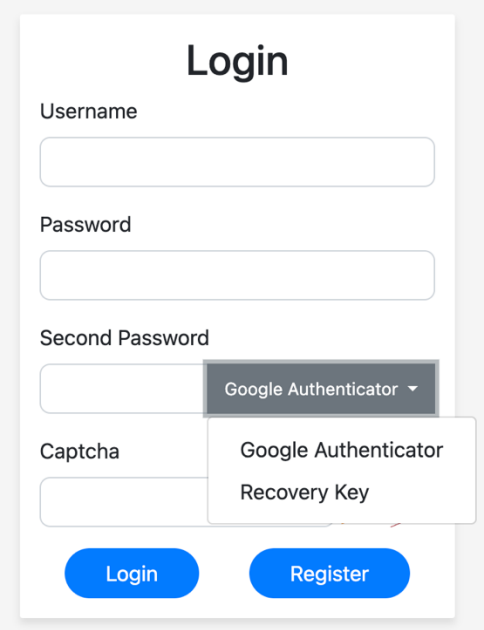
To confirm the user's identity, the password they supply is compared to the password that is kept in the database. "Invalid Username" is the error message that appears if the account does not exist. Verify that the user-supplied password matches the password recorded in the database. The user may be locked if the password is entered incorrectly, increasing the number of failed

attempts. If the password is correct, a return of 'success' indicates that the authentication was successful.

```python
def check_login(userDetails):
    username = userDetails['username']
    # password = userDetails['password']
    password = userDetails['password'].encode('utf-8')  # Encode the password to bytes
    secondpassword = userDetails['secondpassword']
    method = userDetails['method']
    captcha = userDetails['captcha']
    cur = mysql.connection.cursor()
    cur.execute("SELECT user_id,password,limitTime,ratelimit,recoverykey FROM users WHERE username=%s",
                (username,))
    account = cur.fetchone()
    if account[3] >= 5:
        lockuser(username)
        return 'User is been locked'
    if account is None:
        return 'Invalid Username'
    if captcha.lower() != request.cookies.get('captcha').lower():
        error_password(username)
        return 'Invalid Captcha'
    if account[2] is not None and account[2] > datetime.datetime.now():
        return 'User is been locked'
    if account[1] != password:
        error_password(username)
        return 'Invalid Password'
    if not bcrypt.checkpw(password, account[1].encode('utf-8')): # Use bcrypt to check passwords
        error_password(username)
        return 'Invalid Password'
    if method == '0':
        # b32_password = base64.b32encode(password.encode()).decode()
        b32_password = base64.b32encode(password).decode()
        totp = pyotp.TOTP(b32_password)
        result = totp.verify(secondpassword)
        if not result:
            error_password(username)
            return 'Invalid Second Password'
    else:
        if account[4] != secondpassword:
            error_password(username)
            return 'Invalid Second Password'
    session['username'] = username
    session['user_id'] = account[0]
    return 'success'
```

2.1.2 Single-Factor OTP Device (Google Authenticator)

Our implementation uses Google Authenticator to provide a one-time password (OTP) device that enhances security by generating a 6-digit temporary code. There is a drop-down menu in the "second password" section of the Login screen, Allows the selection of Single-Factor OTP Device (Google Authenticator) and Look-Up Secrets (recovery keys) for more stringent login verification.



During the registration phase, a unique key is generated for the OTP and associated with the user account. This key is used to generate OTP, which is then verified at login time.
1. OTP key generation and QR code creation. We used pyotp.random_base32() to generate a random key that will be bound to the user's Google Authenticator app or another compatible OTP app. The provisioning_uri() function creates a QR code that can be scanned by these applications for adding and configuring a user's account.

```python
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.json['username']
        #password = request.json['password']
        password = request.json['password'].encode('utf-8')  # Update: Encode the password to bytes
        recoverykey = request.json['recoverykey']
        # Update: Generate a salt and hash the password
        hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
        cur = mysql.connection.cursor()
        cur.execute("SELECT user_id FROM users WHERE username=%s", (username,))
        account = cur.fetchone()
        if not account:
            if len(password) < 8:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password should be longer than 8 letters'}), 200
            if check_password(password):
                b32_password = base64.b32encode(password.encode()).decode()
                qrcode = pyotp.totp.TOTP(b32_password).provisioning_uri(name=username, issuer_name='WebApp')
                cur = mysql.connection.cursor()
                # cur.execute("INSERT INTO users (username, password,recoverykey,ratelimit) VALUES (%s, %s, %s, %s)", (username, passw
                cur.execute("INSERT INTO users (username, password, secretkey, recoverykey) VALUES (%s, %s, %s, %s)",
                            (username, hashed_password, pyotp.random_base32(), recoverykey))
                mysql.connection.commit()
                return jsonify({'status': 'success', 'message': 'Register Successed', "data": qrcode}), 200
            else:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password has been leaked'}), 200
        else:
            return jsonify({'status': 'error', 'message': 'User has registered'}), 200
    return jsonify({'status': 'success', 'message': 'Register Successed'}), 200
```

2. OTP authentication during login. When the user tries to login and selects the OTP method, the check_login function creates a TOTP object with the key associated with the user stored in the database. The function then invokes this object's verify method to confirm that the user's one-

time password entry is accurate. The user will be able to log in if successful authentication has been made. If this fails, 'Invalid Second Password' is returned.

```
if method == '0':
    totp = pyotp.TOTP(account[2])
    result =totp.verify(secondpassword)
    if not result:
      return 'Invalid Second Password'
```

2.1.3 Look-Up Secrets (recovery keys)

A recovery key is a backup authentication method used to regain access to an account when the primary authentication methods, such as password and OTP, are not available.
In the Single-Factor OTP Device (Google Authenticator) section we mentioned that the "second password" of the Login screen allows the user to select two authentication methods, among them are Look-Up Secrets (recovery keys).
The method of use is to generate the recovery key during the registration process and store it securely. During the login process, users can use their recovery key as an alternative to OTP.
1. The recovery key provided by the user during the registration process is also stored in the database. This allows users to use recovery keys to access their accounts if they lose other authentication methods, such as passwords or OTP devices.

```
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.json['username']
        #password = request.json['password']
        password = request.json['password'].encode('utf-8')  # Update: Encode the password to bytes
        recoverykey = request.json['recoverykey']
        # Update: Generate a salt and hash the password
        hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
        cur = mysql.connection.cursor()
        cur.execute("SELECT user_id FROM users WHERE username=%s", (username,))
        account = cur.fetchone()
        if not account:
            if len(password) < 8:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password should be longer than 8 letters'}), 200
            if check_password(password):
                b32_password = base64.b32encode(password.encode()).decode()
                qrcode = pyotp.totp.TOTP(b32_password).provisioning_uri(name=username, issuer_name='WebApp')
                cur = mysql.connection.cursor()
                # cur.execute("INSERT INTO users (username, password,recoverykey,ratelimit) VALUES (%s, %s, %s, %s)", (username, pass
                cur.execute("INSERT INTO users (username, password, secretkey, recoverykey) VALUES (%s, %s, %s, %s)",
                            (username, hashed_password, pyotp.random_base32(), recoverykey))
                mysql.connection.commit()
                return jsonify({'status': 'success', 'message': 'Register Successed', "data": qrcode}), 200
            else:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password has been leaked'}), 200
        else:
            return jsonify({'status': 'error', 'message': 'User has registered'}), 200
    return jsonify({'status': 'success', 'message': 'Register Successed'}), 200
```

1. Verify the recovery key during login. The check_login function verifies the recovery key that the user enters at login with the recovery key that is kept in the database if the user choose to use the recovery key as the authentication mechanism.
secondpassword is the secondpassword entered by the user and can be used as part of two-factor authentication or as a recovery key. The method variable determines the validation method used. If method is '0', use a time-based one-time password (TOTP) for authentication; Otherwise, the secondpassword is checked to see if it matches the recoverykey stored in the database.

If method is not '0', that is, if TOTP is not used for two-factor validation, the code compares account[4] (i.e., recoverykey in the database) to the secondpassword entered by the user. If they do not match, the 'Invalid Second Password' error is returned.

```
if method == '0':
    # b32_password = base64.b32encode(password.encode()).decode()
    b32_password = base64.b32encode(password).decode()
    totp = pyotp.TOTP(b32_password)
    result = totp.verify(secondpassword)
    if not result:
        error_password(username)
        return 'Invalid Second Password'
else:
    if account[4] != secondpassword:
        error_password(username)
        return 'Invalid Second Password'
session['username'] = username
session['user_id'] = account[0]
return 'success'
```

Memorized Secret Verifiers (§4.2.2 and §5.1.1.2)

Our project complies with the requirements of NIST Special Publication 800-63B, §4.2.2 and §5.1.1.2.

§5.1.1.2: "Memorized secrets SHALL be salted and hashed using a suitable one-way key derivation function". This requirement is satisfied in the register() function of @app.route('/register', methods=['POST']). Here, the password provided by the user during the registration process is salted and hashed using the bcrypt library, and bcrypt is an appropriate one-way key derivation function.

We first need to add the latest version of bcrypt to requirements.txt.

```
Flask==3.0.2
gunicorn==21.2.0
Flask-MySQLdb==2.0.0
Flask-Session==0.6.0
cryptography==42.0.5
PyYAML==6.0.1
Flask-SocketIO==5.3.6
eventlet==0.35.2

pyotp==2.9.0
flask-turnstile==0.1.1
pillow==10.2.0
bcrypt==5.1.1
```

And import bcrypt library in app.py.

```
import bcrypt
```

In the code we implement below, after processing the function bcrypt.gensalt() and bcrypt.hashpw(). (Here is the note: The password that has been processed will be kept in the database; the original password will not be kept.)

```python
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.json['username']
        #password = request.json['password']
        password = request.json['password'].encode('utf-8')  # Update: Encode the password to bytes
        recoverykey = request.json['recoverykey']
        # Update: Generate a salt and hash the password
        hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
        cur = mysql.connection.cursor()
        cur.execute("SELECT user_id FROM users WHERE username=%s", (username,))
        account = cur.fetchone()
        if not account:
            if len(password) < 8:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password should be longer than 8 letters'}), 200
            if check_password(password):
                b32_password = base64.b32encode(password.encode()).decode()
                qrcode = pyotp.totp.TOTP(b32_password).provisioning_uri(name=username, issuer_name='WebApp')
                cur = mysql.connection.cursor()
                # cur.execute("INSERT INTO users (username, password,recoverykey,ratelimit) VALUES (%s, %s, %s, %s)", (username, pass
                cur.execute("INSERT INTO users (username, password, secretkey, recoverykey) VALUES (%s, %s, %s, %s)",
                            (username, hashed_password, pyotp.random_base32(), recoverykey))
                mysql.connection.commit()
                return jsonify({'status': 'success', 'message': 'Register Successed', "data": qrcode}), 200
            else:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password has been leaked'}), 200
        else:
            return jsonify({'status': 'error', 'message': 'User has registered'}), 200
    return jsonify({'status': 'success', 'message': 'Register Successed'}), 200
```

Using Pwned Passwords API to check whether the password was already leaked. And the passwords need to also meet §4.2.2.
On the code side, the check_password function is implemented to prevent users from using leaked passwords.

1. It first checks if the password is fewer than eight characters.
2. If it is, it then uses the SHA-1 hashing method to hash the password,
3. and compares the result with the API to determine if it is stored in the hacked password database.
4. Return True if the password has been leaked, False otherwise.

```python
def check_password(password):
    if len(password) < 8: # Check if the length of the password is less than 8 characters.
        return True
    # Hash the password using SHA-1 encryption and convert it to uppercase.
    sha1_password = hashlib.sha1(password.encode('utf-8')).hexdigest().upper()

    # Split the hashed password into two parts: the first 5 characters (prefix) and the rest (suffix).
    prefix = sha1_password[:5]
    suffix = sha1_password[5:]

    # Construct the API URL using the prefix to check if the password has been exposed in data breaches.
    api_url = f"https://api.pwnedpasswords.com/range/{prefix}"
    response = requests.get(api_url) # Send a GET request to the API URL.

    if response.status_code == 200:
        hashes = response.text.splitlines()

        for hash in hashes:
            if suffix in hash:
                return True
        return False

    # Raise an exception if the API request did not succeed.
    else:
        raise Exception("Failed to check password against API.")
```

Rate-limiting mechanisms (§5.2.2)

To achieve this, we define the lockuser and error_password functions to enforce restrictions on incorrect password attempts. These functions update user records in the database, set time limits, and increase the number of error attempts to meet the requirements of §5.2.2: Implement rate-limiting mechanisms to help reduce the probability that an online attack guesses a user's password by automated means.

1. First, we see that successive failed login attempts are tracked in the check_login function. This function checks to determine if the User should be locked. If the limitTime field of the user is not empty and is greater than the current time, the function returns 'User is been locked', indicating that the user is temporarily unable to log in.

2. account[3] represents the number of failed login attempts associated with the user account in the database. If the count is greater than or equal to 5, the lockuser function is called to lock the

user account.

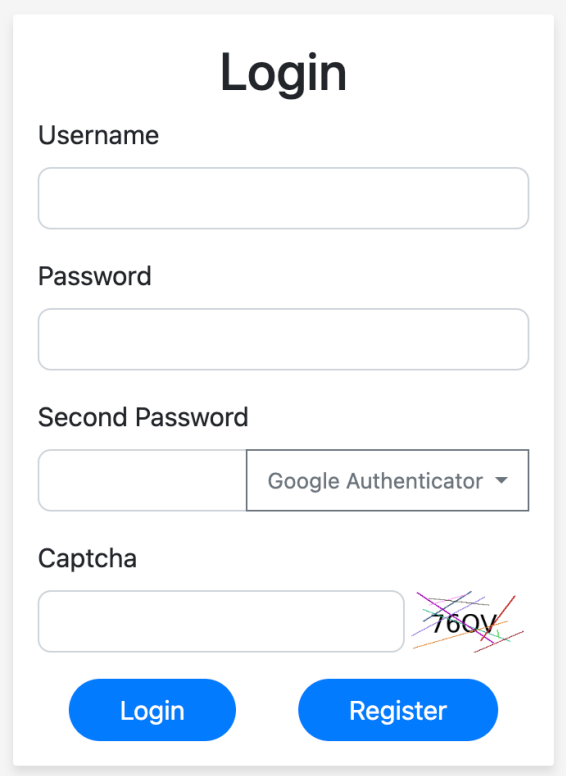
```python
def check_login(userDetails):
    username = userDetails['username']
    # password = userDetails['password']
    password = userDetails['password'].encode('utf-8')  # Encode the password to bytes
    secondpassword = userDetails['secondpassword']
    method = userDetails['method']
    captcha = userDetails['captcha']
    cur = mysql.connection.cursor()
    cur.execute("SELECT user_id,password,limitTime,ratelimit,recoverykey FROM users WHERE username=%s",
                (username,))
    account = cur.fetchone()
    if account[3] >= 5:
        lockuser(username)
        return 'User is been locked'
    if account is None:
        return 'Invalid Username'
    if captcha.lower() != request.cookies.get('captcha').lower():
        error_password(username)
        return 'Invalid Captcha'
    if account[2] is not None and account[2] > datetime.datetime.now():
        return 'User is been locked'
    if account[1] != password:
        error_password(username)
        return 'Invalid Password'
    if not bcrypt.checkpw(password, account[1].encode('utf-8')): # Use bcrypt to check passwords
        error_password(username)
        return 'Invalid Password'
    if method == '0':
        # b32_password = base64.b32encode(password.encode()).decode()
        b32_password = base64.b32encode(password).decode()
        totp = pyotp.TOTP(b32_password)
        result = totp.verify(secondpassword)
        if not result:
            error_password(username)
            return 'Invalid Second Password'
    else:
        if account[4] != secondpassword:
            error_password(username)
            return 'Invalid Second Password'
    session['username'] = username
    session['user_id'] = account[0]
    return 'success'
```

1. The lockuser function that locks user accounts has a field named limitTime that locks user accounts for 5 minutes. This field is checked when the user tries to log in. If the current time is less than limitTime, the user cannot log in.
2. After each login failure, error_password is called and the value of the ratelimit field is increased by 1 to record the number of consecutive failed login attempts.

```python
def lockuser(username):
    # Set the lock time to the current time plus 5 minutes
    limit_time = datetime.datetime.now() + datetime.timedelta(minutes=5)
    cur = mysql.connection.cursor()
    # Update the lock time in the database
    cur.execute("UPDATE users SET limitTime=%s,ratelimit=0 WHERE username=%s", (limit_time,username,))
    mysql.connection.commit()

def error_password(username):
    cur = mysql.connection.cursor()
    # Increase the number of failed attempts
    cur.execute("UPDATE users SET ratelimit=ratelimit+1 WHERE username=%s", (username,))
    mysql.connection.commit()
```

Image-based CAPTCHAs (§5.2.2)

This section will cover the integration of the project into a web application framework that uses Python Flask for back-end processing and HTML/CSS for front-end display. The implementation details cover the generation of capTCHA images, the processing of user input, and the verification process, which is a robust defense against a range of automated attacks.



To implement Image-based CAPTCHAs (§5.2.2), we used Flask's url_for feature to dynamically reference the static resource file captcha.png. This image is generated on the server side and stored in the static file directory.

The user must enter the capTCHA they see in an input field, which uses the required attribute to ensure that the user must fill in the captCHA before submitting the form.

```
<div class="form-group">
    <label for="captcha">Captcha</label>
    <div class="captcha">
        <input type="captcha" class="form-control" id="captcha" name="captcha" required>
        <img src="{{ url_for('static', filename='captcha.png') }}" alt="captcha" width="100px"
            height="38px">
    </div>
</div>
```

In the back-end implementation, the generate_captcha() function is used to generate a 4-digit CAPtCHA consisting of random letters and numbers. This captCHA is unique to the session and is regenerated each time the login page is requested.

Next draw the captcha image using draw_random_line and create_captcha_image, create an image using Python's PIL library, and then draw the captcha text onto the image. In addition, random lines are drawn to increase the difficulty of identification and prevent automated tools from easily recognizing the captCHA.

```
def generate_captcha():
    code = ''
    for _ in range(4):
        code += random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890')
    return code

def draw_random_line(draw, width, height):
    start = (random.randint(0, width), random.randint(0, height))
    end = (random.randint(0, width), random.randint(0, height))
    color = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))
    draw.line([start, end], fill=color, width=2)

def create_captcha_image(code):
    image = Image.new('RGB', (200, 100), color = (255, 255, 255))
    font = ImageFont.truetype('DejaVuSans.ttf', 40)
    d = ImageDraw.Draw(image)
    d.text((40, 30), code, fill=(0, 0, 0), font=font)
    for _ in range(10):
        draw_random_line(d, image.width, image.height)
    image.save('static/captcha.png')
```

Finally, the verification code that validates user input. It uses two functions to handle the login and validate the capTCHA entered by the user. When the user submits the login form, the back-end verifies that the verification code entered is the same as the verification code stored in the cookie (case insensitive). If not, an error message indicating that the verification code is invalid is returned.
The login function is used to process the user's login request, verify the content submitted by the form, generate, and send a verification code cookie, and redirect to the home page on success or

return to the login page on failure.

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        error=check_login(request.form)
        if error=='success':
            return redirect(url_for('index'))
    code = generate_captcha()
    create_captcha_image(code)
    resp = make_response(render_template('login.html',error=error))
    resp.set_cookie('captcha', code)
    return resp
```

The check_login function is used to validate login credentials, check that the user's identity and the verification code entered are correct, and return the appropriate error message if the user's account is locked or the credentials are invalid.

```python
def check_login(userDetails):
    username = userDetails['username']
    # password = userDetails['password']
    password = userDetails['password'].encode('utf-8')  # Encode the password to bytes
    secondpassword = userDetails['secondpassword']
    method = userDetails['method']
    captcha = userDetails['captcha']
    cur = mysql.connection.cursor()
    cur.execute("SELECT user_id,password,limitTime,ratelimit,recoverykey FROM users WHERE username=%s",
                (username,))
    account = cur.fetchone()
    if account[3] >= 5:
        lockuser(username)
        return 'User is been locked'
    if account is None:
        return 'Invalid Username'
    if captcha.lower() != request.cookies.get('captcha').lower():
        error_password(username)
        return 'Invalid Captcha'
```

Create new account and bind authenticators (OTP and recovery keys)

Our project provides the ability to bind authenticators (OTP and recovery keys) when registering a new account.
The register() function first checks if a user with the same name already exists in the database. If it does not exist, the password is checked for security (the check_password function), then the OTP key (secretkey) is generated, and a two-dimensional code (qrcode) compatible with the OTP application is created. Finally, the username, password, OTP key, and recovery key are stored in the database together. If registration is successful, a JSON response is returned with a

message of successful registration and QR code data.

```python
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        # Extract 'username', 'password' and 'recoverykey' from the JSON body of the request.
        username = request.json['username']
        password = request.json['password']
        recoverykey = request.json['recoverykey']
        cur = mysql.connection.cursor() # Create a new cursor to interact with the MySQL database.
        cur.execute("SELECT user_id FROM users WHERE username=%s", (username,)) # Execute a SQL query to check if a user with the given username exists.
        account = cur.fetchone() # Fetch the first row of the results from the SQL query.
        if (not account): # Check if the account does not exist.
            result =check_password(password)
            if (not result):
                secretkey =pyotp.random_base32() # Generate a random secret key for TOTP authentication.
                # Create a TOTP URI for provisioning using the secret key and username.
                qrcode= pyotp.totp.TOTP(secretkey).provisioning_uri(name=username, issuer_name='WebApp')
                cur = mysql.connection.cursor()
                # Insert the new user details into the 'users' table.
                cur.execute("INSERT INTO users (username, password, secretkey,recoverykey) VALUES (%s, %s, %s, %s)", (username, password, secretkey,recoverykey))
                mysql.connection.commit()
```

Corresponding to the registration form of the front-end login.html, the user needs to enter the username, password, and the second password for recovery of the key, the second password is recovery keys. When the user successfully registers, a one-time random QR code will be generated automatically according to the user's secret key. the user needs to scan the QR code with Google authenticators to obtain the OTP (one-time password) and bind the new account and authenticators at the same time. To test this implementation, a drop-down menu is included in the second password cell of the user's login interface, allowing the user to choose between using Google Validator or recovery key for secondary authentication.

```html
<div class="register-container">
    <h2 class="text-center">Register</h2>
    <form>
        <div class="form-group">
            <label for="register-username">Username</label>
            <input type="text" class="form-control" id="register-username" name="register-username"
                autocomplete="false">
        </div>
        <div class="form-group">
            <label for="register-password">Password</label>
            <input type="password" class="form-control" id="register-password" name="register-password"
                autocomplete="false">
        </div>
        <div class="form-group">
            <label for="register-second-password">Second Password</label>
            <input type="password" class="form-control" id="register-second-password"
                name="register-second-password" autocomplete="false">
        </div>
        <div class="text-center" style="display: flex;justify-content: space-around;">
            <button type="button" class="btn btn-primary" onclick="register()">Register</button>
            <button type="button" class="btn btn-primary" onclick="GoToLogin()">Go to Login</button>
        </div>
    </form>
</div>
```

In JavaScript, the register() function takes care of the registration logic, including collecting the username, password, and recovery code from the form and sending it to the server. If registration is successful, a QR code (for OTP authentication) will be displayed.

```
function register() { // Registration logic (collecting data and sending requests)
    var username = document.getElementById('register-username').value;
    var password = document.getElementById('register-password').value;
    var recoverykey = document.getElementById('register-second-password').value;

    // Front-end validation, simply checking that the input field is empty in the register() function:
    if (username === '' || password === '' || recoverykey === '') {
        alert('Please fill in all fields');
        return;
    }
    // Form Submission
    fetch('/register', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            username: username,
            password: password,
            recoverykey: recoverykey
        })
    }).then(response => {
        if (response.ok) {
            response.json().then(data => {
                if (data.status == 'success') { // QRcode

                    document.querySelector('.register-container').style.display = 'none';
                    document.querySelector('.qrcode-container').style.display = 'block';
                    var qrcode = new QRCode(document.querySelector('.qrcode'), {
                        text: data.data,
                        width: 200,
                        height: 200
                    });
                }
                else {
                    alert(data.message);
                }
            });
        } else {
            response.json().then(data => {
                alert(data.error);
            });
        }
    });
}
```

When a user submits registration information, the front end sends a request to the back-end /register route, which handles the registration logic, including creating an account and binding an authenticator.

Proper session binding (§7.1)

NIST Special Publication 800-63B: §7.1 emphasizes the importance of session binding, specifically requiring that appropriate session management measures be implemented to ensure

that sessions cannot easily be hijacked and that the user's identity can be securely associated with the session.

app.py uses the session object of Flask framework for session management and uses the file system to store session data, which ensures the security and correct management of session data and meets the requirements of session binding.
And the session signature (SESSION_USE_SIGNER) is configured, which adds extra security because it ensures that the session cookie is signed and not tampered with.

```
app = Flask(__name__)
# Configure secret key and Flask-Session
app.config['SECRET_KEY'] = 'your_secret_key_here'
app.config['SESSION_TYPE'] = 'filesystem'  # Options: 'filesystem', 'redis', 'memcached', etc.
app.config['SESSION_PERMANENT'] = False
app.config['SESSION_USE_SIGNER'] = True  # To sign session cookies for extra security
app.config['SESSION_FILE_DIR'] = './sessions'  # Needed if using filesystem type

# Initialize the Flask-Session
Session(app)
publicKey = {}
```

When the user successfully logs in, the check_login function returns 'success', and session information such as the user's ID or other identifier is stored in the Flask session. In addition, session['user_id'] and session['username'] are used to store user session information, ensuring that the session is bound to the user.

```
session['username'] = username
session['user_id'] = account[0]
```

When processing a request, checking for the presence of a 'user_id' in the session is a way of confirming that the user is logged in.

```
@app.route('/send_message', methods=['POST'])
def send_message():
    if not request.json or not 'ciphertext' in request.json or not 'iv' 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)
```

This login function handles user login requests. After a successful login, the check_login function verifies the username, password, second password, and verification code. Once the authentication is successful, make_response and set_cookie will be used to set the cookie, and the user ID and other necessary information will be stored in the session, implementing the

binding between the session and the user.

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        error=check_login(request.form)
        if error=='success':
            return redirect(url_for('index'))
    code = generate_captcha()
    create_captcha_image(code)
    resp = make_response(render_template('login.html',error=error))
    resp.set_cookie('captcha', code)
    return resp
```

3. E2EE

3.1

```javascript
async function initialization() {
    let ecKeys = await crypto.subtle.generateKey({name: "ECDH", namedCurve: "P-384"}, true, ["deriveKey"],);
    let jwk_PublicKey = await crypto.subtle.exportKey("jwk", ecKeys.publicKey);
    let jwk_PrivateKey = await crypto.subtle.exportKey("jwk", ecKeys.privateKey);

    const response = await fetch('/push_public_key', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({posterID:myID, publicKey: jwk_PublicKey}),
    });

    const data = await response.json()

    console.log(data.message)
    localStorage.setItem("publicKey",JSON.stringify(jwk_PublicKey));
    localStorage.setItem("privateKey", JSON.stringify(jwk_PrivateKey));
    console.log("Public key: ", localStorage.getItem("publicKey"));
    console.log("Private key: ", localStorage.getItem("privateKey"));

    if (localStorage.getItem(`ivCounters_${myID}to${peer_id}`) !== null){
        ivCounters = parseInt(localStorage.getItem(`ivCounters_${myID}to${peer_id}`))
    }else{
        // ivCounters = 0
    }

}
```

We call the "crypto.subtle.generateKey" method in the WebCrypto API to generate the ECDH public key and private key, and specify the use of P-384 elliptic curve (the elliptic curve indicated in the requirements should be used). Export the public key to JWK format and send it to the server through a post type request.

```
async function pull_peerPublicKey() {
    if (peer_id === -1) {
        console.error("No peer selected to request public key for!");
        return;
    }
    if (localStorage.getItem(`peerPK_${peer_id}`) !== null){
        console.log(`peer publickey of user ${peer_id} already stored: `,localStorage.getItem(`peerPK_${peer_id}`));
        return;
    }

    try{
        const response = await fetch(`/pull_public_key/${peer_id}`,{
            method: 'GET',
            heater:{
                'Accept':'application/json',
            }
        });

        if (!response.ok){
            const data = await response.json();
            console.error("Error in HTTP:",data.message)
        }

        const { publicKey } = await response.json();
        const peerPublicKey = await crypto.subtle.importKey(
                "jwk",
                publicKey,
                {
                    name: "ECDH",
                    namedCurve: "P-384"
                },
                true,
                [],
        );
        let jwk_peerPublicKey = await crypto.subtle.exportKey("jwk", peerPublicKey);
        localStorage.setItem(`peerPK_${peer_id}`, JSON.stringify(jwk_peerPublicKey));
        console.log("Peer publickey obtained successfully: ", localStorage.getItem(`peerPK_${peer_id}`));
        const AES_MAC = await deriveKeys(peerPublicKey);
```

The specific route on the backend receives and stores it. Then also send a post type request to obtain the other party's public key. The backend specific route returns the public key to the frontend. Generate a shared key based on ECDH key exchange through your own private key and the other party's public key.

3.2

```
if (sharedSecret) {
    console.log("Shared secret derived successfully", sharedSecret)
    const AESOptions = {
        "algorithm":
            {
                name: "HKDF",
                hash: "SHA-256",
                salt: numToUint8Array(salt),
                info: new Uint8Array(`CHAT_KEY_USER${myID}to${peer_id}`),
            },
        "derivedKeyAlgorithm":
            {
                name: "AES-GCM",
                length: 256,
            },
    }

    const MACOptions = {
        "algorithm":
            {
                name: "HKDF",
                hash: "SHA-256",
                salt: numToUint8Array(salt),
                info: new Uint8Array(`CHAT_MAC_USER${myID}to${peer_id}`),
            },
        "derivedKeyAlgorithm":
            {
                name: "HMAC",
                hash: "SHA-256",
                length: 256,
            },
    }
```

After the shared key is successfully generated, use the HKDF algorithm to derive the AES key and MAC key. In this process we have a sole salt. Through "numToUint8Array(salt)", the value used in each derivation process is different, thereby improving security.
"info" acts as a label in the code. Indicates whether the key is used to encrypt messages from user 1 to user 2 or to verify the integrity of messages from user 1 to user 2. Indicates whether user 1 or user 2 is the starting point.

```
const AESKey = await crypto.subtle.deriveKey(
    AESOptions["algorithm"],
    sharedSecret,
    AESOptions["derivedKeyAlgorithm"],
    true,
    ["encrypt", "decrypt"]
)

const MACKey = await crypto.subtle.deriveKey(
    MACOptions["algorithm"],
    sharedSecret,
    MACOptions["derivedKeyAlgorithm"],
    true,
    ["sign", "verify"]
)
```

The AES key is used to encrypt and decrypt information, and the MAC key is used to ensure the authenticity and integrity of information.

3.3

```
let encoded = new TextEncoder('utf-8').encode(message);
const ciphertext = await window.crypto.subtle.encrypt(
        {
            name: "AES-GCM",
            iv: ivBuffer,
        },
        aesKey,
        encoded
);
```

As shown in the figure above, we use AES-GCM mode for encryption.

```
const ivBuffer = numToUint8Array(ivCounters);
const tag = await window.crypto.subtle.sign(
        "HMAC",
        macKey,
        ivBuffer
);
```

The initialization IV generally defaults to 96 bits (in GCM mode), and we use a counter to ensure that the IV used for each encryption is unique.

```
const payload = {
    receiver_id: peer_id,
    iv: base64Encode(ciphertextBody.iv),
    ciphertext: base64Encode(ciphertextBody.ciphertext),
    tag: base64Encode(ciphertextBody.tag)
};
console.log('payload', JSON.stringify(payload))
fetch('/send_message', {
    method: 'POST', // Specify the method
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify(payload),
```

The image illustrates how we deliver the recipient both the IV and the ciphertext in a JSON format.

```
try {
    const ivBuffer = base64Decode(message.iv)
    const tag = base64Decode(message.tag)

    const isValid = await crypto.subtle.verify(
        "HMAC",
        macKey,
        tag,
        ivBuffer,
    );
    if (isValid) {
        if (message.sender_id === myID && message.receiver_id === peer_id) {
            ivCounters++;
            localStorage.setItem(`ivCounters_${myID}to${peer_id}`, ivCounters)
        }
    }
    return {
        isValid: isValid,
        iv: ivBuffer
    }
} catch (error) {
    console.error('checkIV', error)
    return false
}
```

We use the 'ivCounters++' function to increase the IV. The most recent IV counter value is stored and the ivCounters are only incremented following a successful HMAC verification. Ensures that the number of IVs received increases progressively. Side confirms IV > IV−1 to thwart replay attacks.

```
async function encryptMessage(aesKey, macKey, message, ivCounters) {
    const ivBuffer = numToUint8Array(ivCounters);
    const tag = await window.crypto.subtle.sign(
            "HMAC",
            macKey,
            ivBuffer
    );
```

To prevent the attacker from selecting IVs, use 'window.crypto.subtle.sign' to protect the IV using HMAC-SHA256 while using the MAC key.

3.4

```
localStorage.setItem("publicKey",JSON.stringify(jwk_PublicKey));
localStorage.setItem("privateKey", JSON.stringify(jwk_PrivateKey));
console.log("Public key: ", localStorage.getItem("publicKey"));
console.log("Private key: ", localStorage.getItem("privateKey"));
```

```
let jwk_peerPublicKey = await crypto.subtle.exportKey("jwk", peerPublicKey);
localStorage.setItem(`peerPK_${peer_id}`, JSON.stringify(jwk_peerPublicKey));
console.log("Peer publickey obtained successfully: ", localStorage.getItem(`peerPK_${peer_id}`));
```

```
crypto.subtle.exportKey("jwk", AES_MAC.AESKey).then(jwkAES => {
    localStorage.setItem("AESKey", JSON.stringify(jwkAES))
    console.log("AES Key derived successfully: ", AES_MAC.AESKey);
})

crypto.subtle.exportKey("jwk", AES_MAC.MACKey).then(jwkMAC => {
    localStorage.setItem("MACKey", JSON.stringify(jwkMAC))
    console.log("MAC Key derived successfully: ", AES_MAC.MACKey);
```

As illustrated in the figures above, when the needed key is generated, it is instantly put in local storage using 'localStorage.setItem'. (Keys needs to be converted into a json string before storing).

3.5

(Display the history of previous messages being exchanged + new messages)
For front end:

```
// Fetch messages from server
function fetchMessages() {

    if (peer_id === -1) return; // Exit if no peer selected
    fetch(`/fetch_messages?last_message_id=${lastMessageId}&peer_id=${peer_id}`)
        .then(response => response.json())
        .then(data => {
            data.messages.forEach(message => {
                if (message.second_tag !== null){ // indicating that it is a special key refresh message
                    verifyNewMac(message).then(verifyBody=>{
                        if (verifyBody.isValid) {
                            updateKeys(verifyBody.newKeys)

                            const messagesContainer = document.getElementById('messages');
                            const messageElement = document.createElement('div');

                            messageElement.textContent = `|--------Key Changed--------|`;
                            messagesContainer.appendChild(messageElement);
                            Salt++
                        }
                    })
                }else
                    displayMessage(message);    //since the content of the special message is not encrypted, there is n

                lastMessageId = message.message_id;
            });
        })
        .catch(error => console.error('Error fetching messages:', error));
}
```

The first thing this function does is see if the contact object is chosen. In order to retrieve historical messages, the "/fetch_messages" route in app.py will only be called when "peer_id!== -1". Verify whether "second_tag" is present in the returned data. If not, this indicates that the message is just a typical chat message. The message will then be printed by calling the "displayMessage" function.

```
// Display a single message
async function displayMessage(message) {
    const AESKey = await importKey("AESKey")
    decryptMessage(AESKey, message).then(message_text => {
        if (message_text) {
            const messagesContainer = document.getElementById('messages');
            const messageElement = document.createElement('div');

            // Determine sender and receiver strings
            const sender = message.sender_id === myID ? "me" : (userInfo[message.sender_id] || `User ${message.sender_id}`);
            const receiver = message.receiver_id === myID ? "me" : (userInfo[message.receiver_id] || `User ${message.receiver_

            messageElement.textContent = `From ${sender} to ${receiver}: ${message_text === "change" ? "Key changed":message_t
            messagesContainer.appendChild(messageElement);
        }
    })
}
```

First, call "importKey" in the method "displayMessage" to get the decrypted AES key and transform it from JWK format to a useable format. Next, use the modified AES key to decrypt the message by using "decryptMessage". The sender and recipient are identified once the message has been successfully encrypted. Print the data on the page if the message text is not "change" (not a system message).

For backend:

```python
@app.route('/fetch_messages')
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( key: 'peer_id', type=int)

    cur = mysql.connection.cursor()
    query = """SELECT message_id,sender_id,receiver_id,message_text,iv,tag,second_tag 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})
```

The first thing the application does is see if the user is signed in. Ask to obtain the "peer_id" and "last_message_id" after that. The prior one is the message's last ID that the ID was lastly obtained. We only receive new messages that have not previously been obtained (that is, beginning with the last message after last_message_id). Subsequently, run the SQL query, save the requested data in dictionary format, and send it back to the client.

(If Local Storage has been cleared, prior messages cannot be decrypted; display warning.)

```javascript
setTimeout(() => {
    window.addEventListener('storage', async () => {
        if (localStorage.length === 0 ) {
            alert("Local Storage has been cleared, previous messages cannot be decrypted");
```

Because the local storage is empty when you first reach the page (even though the function "initialization" is executed immediately), the program creates a timeout to avoid displaying an error alert. The software configures a listener on the front end to track the web page's local storage state continuously. There will be a pop-up notice when the local storage is cleared.

3.6

```javascript
async function refreshKeys() {
    if (peer_id === -1) return; // Exit if no peer selected

    // const Old_AESKey = await importKey("AESKey")
    const Old_MACKey = await importKey("MACKey",Salt)
    const PeerPublicKey = await importKey(`peerPK`)
```

First call the ‘importKey’ function to import ‘Old_MACKey’ (current MAC key).
And we import another user's public key and use it to derive a shared secret from user’s own private key.

```javascript
const new_Keys = await deriveKeys(PeerPublicKey,Salt+1)
const message = new TextEncoder('utf-8').encode("change")
const ivBuffer = numToUint8Array(ivCounter)
```

Call the 'deriveKeys' function, use 'salt+1' to ensure that the salt used is different each time, and then generate new AES key and MAC key.

Encode 'change' into UTF-8 format to notify that the key is about to be changed. Then use ‘numToUint8Array’ to generate a new IV value (guarantee that the IV used is new every time).

```
const old_tag = await window.crypto.subtle.sign(
    "HMAC",
    Old_MACKey,
    ivBuffer
);

const new_tag = await window.crypto.subtle.sign(
    "HMAC",
    new_Keys.MACKey,
    ivBuffer
);
```

We need to use the old MAC key and the new MAC key to sign the same IV separately to generate two tags. The operation meets the project's requirement which requires that we need to use both the old mac key and the new mac key to protect messages.

```
const payload = {
    receiver_id: peer_id,
    iv: base64Encode(ivBuffer),
    ciphertext: base64Encode(message),
    tag: base64Encode(old_tag),
    second_tag: base64Encode(new_tag)
};

console.log('payload', JSON.stringify(payload))

fetch('/refreshKey', {
    method: 'POST', // Specify the method
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify(payload),
}).then(response => {
    if (!response.ok) {
        // If the server response is not OK, throw an error
        throw new Error('Network response was not ok');
    }
    return response.json(); // Parse JSON response from the server
}).then(data => {
    console.log(data.status)
    console.log(`Old Mac: ${Old_MACKey}\nNew Mac: ${new_Keys.MACKey}`)
    document.getElementById('messageInput').value = ''; // Clear the input after sending
```

We send the message to the server through a ‘post’ type http request. As can be seen from the above code, the message sent contains iv, "change" string, two tags and other data required in the project.

```python
@app.route( rule: '/refreshKey', methods=['POST'])
def refreshKey():
    if 'user_id' not in session:
        abort(403)

    if not request.json or not ('iv' in request.json and 'tag' in request.json and 'second_tag' in request.json):
        abort(400)

    sender_id = session['user_id']
    receiver_id = request.json['receiver_id']
    message_text = request.json['ciphertext']
    iv = request.json['iv']
    tag = request.json['tag']
    second_tag = request.json['second_tag']

    save_message(sender_id, receiver_id, message_text, iv, tag, second_tag)

    print("Server received a request for refreshKey")
    return jsonify({'status': 'success', 'message': 'Keys changed'}), 200
```

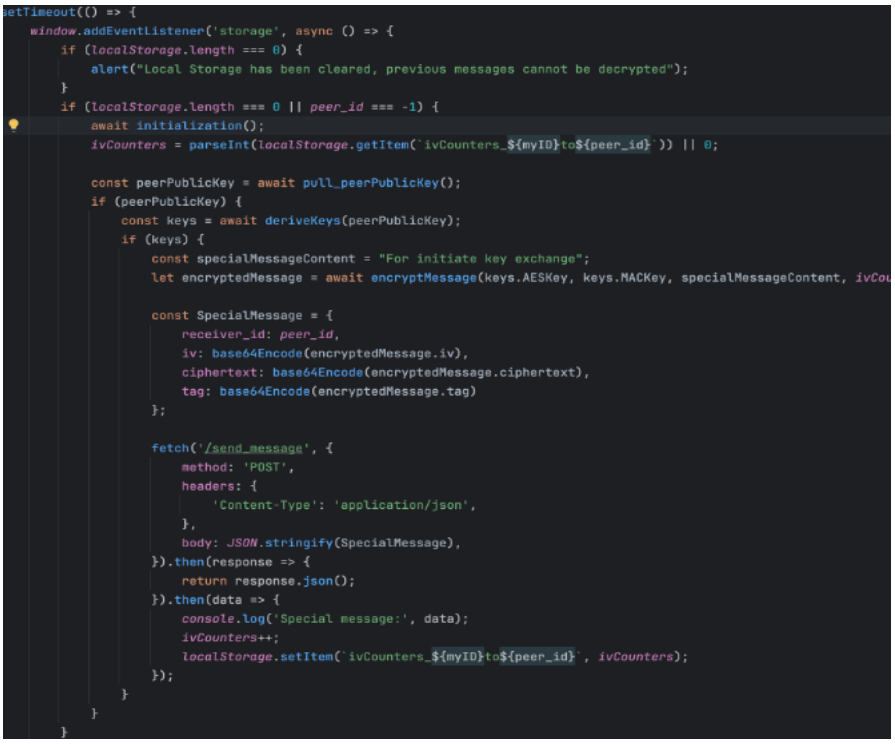
The backend has a specific route for receiving this request. We firstly uses the stored old MAC key to verify the received IV and message content, and checks whether the ‘old_tag’ is valid. After the old MAC key is successfully verified, use the new MAC key to verify the received message and check whether the ‘new_tag’ is valid. When both verifications are valid, "Key has changed" is displayed on the front end.

```python
2 usages
def save_message(sender, receiver, ciphertext, iv, tag, second_tag=None):
    cur = mysql.connection.cursor()
    if second_tag is None:
        cur.execute("INSERT INTO messages (sender_id, receiver_id, message_text,iv,tag) VALUES (%s, %s, %s, %s, %s)",
                    (sender, receiver, ciphertext, iv, tag))
    else:
        cur.execute(
            "INSERT INTO messages (sender_id, receiver_id, message_text,iv,tag, second_tag) "
            "VALUES (%s, %s, %s, %s, %s, %s)", (sender, receiver, ciphertext, iv, tag, second_tag))
    mysql.connection.commit()
    cur.close()
```

The backend has a dedicated route to verify whether it means that the message was sent by clicking the refresh key. If so, it is stored in the database.

3.7

For front end:



```
setTimeout(() => {
    window.addEventListener('storage', async () => {
        if (localStorage.length === 0) {
            alert("Local Storage has been cleared, previous messages cannot be decrypted");
        }
        if (localStorage.length === 0 || peer_id === -1) {
            await initialization();
            ivCounters = parseInt(localStorage.getItem(`ivCounters_${myID}to${peer_id}`)) || 0;

            const peerPublicKey = await pull_peerPublicKey();
            if (peerPublicKey) {
                const keys = await deriveKeys(peerPublicKey);
                if (keys) {
                    const specialMessageContent = "For initiate key exchange";
                    let encryptedMessage = await encryptMessage(keys.AESKey, keys.MACKey, specialMessageContent, ivCou

                    const SpecialMessage = {
                        receiver_id: peer_id,
                        iv: base64Encode(encryptedMessage.iv),
                        ciphertext: base64Encode(encryptedMessage.ciphertext),
                        tag: base64Encode(encryptedMessage.tag)
                    };

                    fetch('/send_message', {
                        method: 'POST',
                        headers: {
                            'Content-Type': 'application/json',
                        },
                        body: JSON.stringify(SpecialMessage),
                    }).then(response => {
                        return response.json();
                    }).then(data => {
                        console.log('Special message:', data);
                        ivCounters++;
                        localStorage.setItem(`ivCounters_${myID}to${peer_id}`, ivCounters);
                    });
                }
            }
        }
```

Use a listener to track the local storage status of a web page in real time. If the local storage is cleared or the shared secret is removed, restart the ECDH key exchange. To generate the user's public and private keys, call "initialization" first. Use the function "pull_peerPublicKey" to retrieve the public key of the other party from the server. Next, use the function "deriveKeys" to derive the MAC and AES keys. Create a unique message and use "encryptMessage" for encryption after creating a new key. Convert the ciphertext to JSON format before sending it to the server and updating "ivCounters".

For back end:

```python
@app.route( rule: '/send_message', methods=['POST'])
def send_message():
    if not request.json or not 'ciphertext' in request.json or not 'iv' 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)

    sender_id = session['user_id']
    receiver_id = request.json['receiver_id']
    message_text = request.json['ciphertext']
    iv = request.json['iv']
    tag = request.json['tag']

    save_message(sender_id, receiver_id, message_text, iv, tag)

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


2 usages
def save_message(sender, receiver, ciphertext, iv, tag, second_tag=None):
    cur = mysql.connection.cursor()
    if second_tag is None:
        cur.execute("INSERT INTO messages (sender_id, receiver_id, message_text,iv,tag) VALUES (%s, %s, %s, %s, %s)"
                    (sender, receiver, ciphertext, iv, tag))
    else:
        cur.execute(
            "INSERT INTO messages (sender_id, receiver_id, message_text,iv,tag, second_tag) "
            "VALUES (%s, %s, %s, %s, %s, %s)", (sender, receiver, ciphertext, iv, tag, second_tag))
    mysql.connection.commit()
    cur.close()
```

Check the message and take out important components such the tag, iv, and ciphertext. To save the message in the database, call ‘save_message’ (which includes the SQL link).

3.8

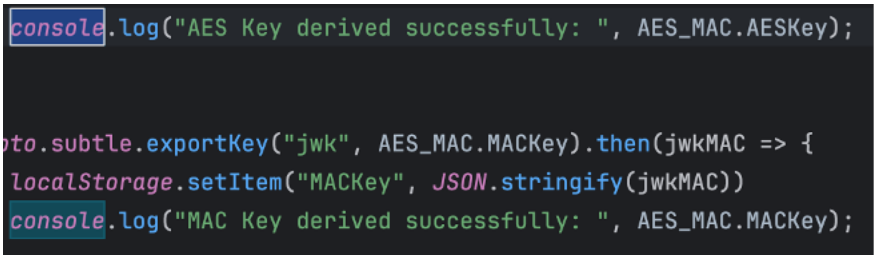
```javascript
async function encryptMessage(aesKey, macKey, message, ivCounters) {
    const ivBuffer = numToUint8Array(ivCounters);
    const tag = await window.crypto.subtle.sign(
            "HMAC",
            macKey,
            ivBuffer
    );

    let encoded = new TextEncoder('utf-8').encode(message);
    const ciphertext = await window.crypto.subtle.encrypt(
```

All chat messages will be encoded using UTF-8, as you can see in the ‘encrypteMessage’ function.

```javascript
    const payload = {
        receiver_id: peer_id,
        iv: base64Encode(ivBuffer),
        ciphertext: base64Encode(message),
        tag: base64Encode(old_tag),
        second_tag: base64Encode(new_tag)
    };

    console.log('payload', JSON.stringify(payload))

    fetch('/refreshKey', {
        method: 'POST', // Specify the method
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(payload),
```

Every network data sent between users will be stored in JSON format. The code snapshot above also illustrates our unique schema ({"ciphertext":"…", "IV":"…"}).

3.9



Here are a few instances in the images above. All of the necessary data is printed in the log by our program.

4. TLS

4.1

```
ssl_protocols TLSv1.3;
```

Let Nginx only support TLS 1.3 protocol.

4.2

```
ssl_ecdh_curve X25519:P-384;
```

Specifies Nginx to use P-384 in the X25519 elliptic group in the TLS handshake.

4.3

```
ssl_prefer_server_ciphers on;
ssl_ciphers ECDHE-ECDSA-CHACHA20-POLY1305;
ssl_conf_command Options PrioritizeChaCha;
ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256;
```

Use the 'PrioritizeChaCha' command and the 'Ciphersuites' command to ensure that Nginx only uses the ECDHE-ECDSA-CHACHA20-POLY1305 cipher suite.

4.4

```
ssl_stapling off;
ssl_stapling_verify off;
```

Because we need to use our self-signed CA certificate, so we need to make sure no OCSP stappling.

4.5

```
add_header Strict-Transport-Security "max-age=604800; includeSubDomains" always;
```

Set up HSTS and keep it running for a week. During this week, all communication must be done through https.

4.6

Our certificate meets all the requirements below:

1. X.509 version 3
2. ECDSA public key over P-384
3. SHA384 as hashing algorithm for signature
4. CA flag (critical): false
5. Key Usage (critical) = Digital Signature
6. Extended Key Usage = Server Authentication
7. Include both Subject Key Identifier and Authority Key Identifier
8. Validity period = 90 days

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            2d:3d:94:ce:45:52:63:20:1c:92:1f:49:15:c6:8a:ee:0b:34:2b:e5
        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  3 09:25:03 2024 GMT
            Not After : Jul  2 09:25:03 2024 GMT
        Subject: C = HK, CN = group-31.comp3334.xavier2dc.fr
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:d8:6d:c2:0c:37:13:03:11:2a:3b:e1:2f:96:d3:
                    c1:18:26:51:75:61:0b:cf:a3:81:6a:c9:3f:c9:eb:
                    61:fc:cc:8a:db:62:86:b5:2f:26:c0:56:9d:f8:2b:
                    a3:86:16:c0:78:d4:d3:fc:8f:26:e0:85:2b:6b:0e:
                    d2:a0:af:04:38:eb:43:65:a9:c6:f3:56:f2:60:1d:
                    9f:dc:2a:51:1b:15:9b:67:f7:86:1e:7d:b6:85:6b:
                    9e:e4:c4:db:a5:0e:8c
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:group-31.comp3334.xavier2dc.fr
            X509v3 Subject Key Identifier:
                F8:EB:F4:7A:B7:0F:33:A7:17:93:5C:79:EE:A8:8E:0C:B9:D0:FC:95
            X509v3 Authority Key Identifier:
                keyid: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
    Signature Algorithm: ecdsa-with-SHA384
         30:81:87:02:41:2d:a9:08:26:95:c3:e2:ec:c2:56:59:7d:12:
         d0:54:5a:3b:2a:8b:94:74:1e:d8:33:d9:13:74:74:ec:5e:78:
         4a:86:78:1a:e1:ea:90:7c:2f:ce:65:90:5b:e2:21:e7:8e:05:
         4a:d5:5c:7a:63:5f:3b:5e:bb:77:b1:9f:17:a9:81:46:02:42:
         01:24:01:d6:bc:c8:b5:26:1b:0d:9b:8f:3a:02:76:64:13:5d:
         ef:be:7a:70:c2:28:b9:65:41:f7:5d:17:af:df:21:af:d6:e2:
         dc:54:8d:35:0d:ae:bb:89:1a:00:7f:f3:1d:fd:57:68:cc:4f:
         32:c7:ff:ba:6d:9a:ea:10:d8:d0:04:c2
```

90 days

4.7

```
server {
    listen 8443 ssl http2;
    server_name group-31.comp3334.xavier2dc.fr;
```

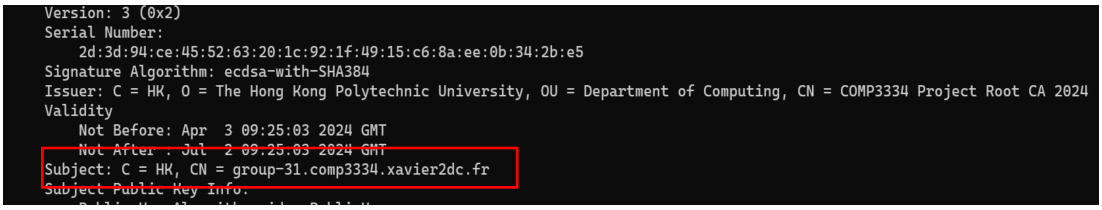
We are group 31. The website should be hosted at:
'https://group 31.comp3334.xavier2dc.fr:8443/'

4.8

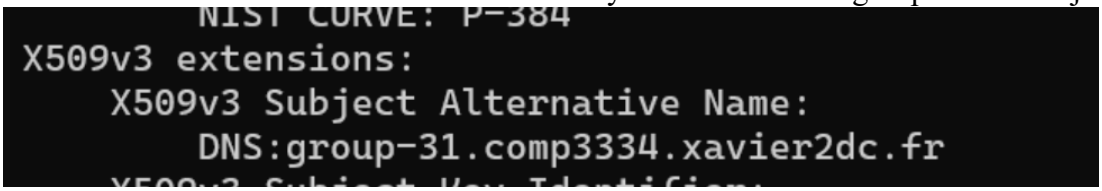
```
# localhost name resolution is handled within DNS itself.
#	127.0.0.1	localhost
#	::1		localhost

localhost group-31.comp3334.xavier2dc.fr

# Added by Docker Desktop
10.11.67.97 host.docker.internal
10.11.67.97 gateway.docker.internal
# To allow the same kube context to work on the host and the container:
127.0.0.1 kubernetes.docker.internal
# End of section
```

We can use 'group31.comp3334.xavier2dc.fr' directly instead of 'localhost'.

4.9



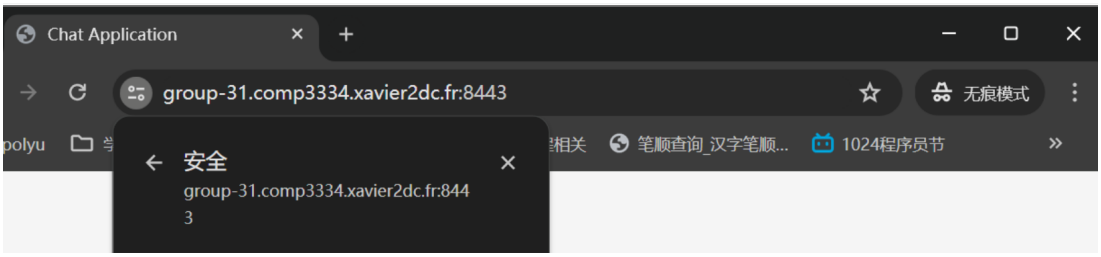
We can observe that the domain name is already connected to our group in the "subject" field.



The domain name appears in both ‘common name’ and ‘subject alternative name’.

4.10

We limited the domain of the certificate to a subdomain of comp3334.xavier2dc.fr, as you can see from the screenshot titled "4.6."



The certificate is truly trusted and secure.

5. Explanation of the Solution

5.1 Storage and Verification of User Passwords

Our project uses the method of user password storage and verification in the Web application built by Flask framework and MySQL database. The application implements registration and login functions, including secure storage of passwords and accurate verification during login. We used several Technology Stack including Flask, MySQL, bcrypt, pyotp, and Pillow (PIL Fork).

```
Flask==3.0.2
gunicorn==21.2.0
Flask-MySQLdb==2.0.0
Flask-Session==0.6.0
cryptography==42.0.5
PyYAML==6.0.1
Flask-SocketIO==5.3.6
eventlet==0.35.2

pyotp==2.9.0
flask-turnstile==0.1.1
pillow==10.2.0
bcrypt==5.1.1
```

In order to ensure that passwords stored in the database do not exist in plain text during the user registration process, we use the bcrypt library to encrypt passwords. bcrypt is an adaptive hash function based on Blowfish ciphers that enhances the security of password storage by salting and hashing it multiple times.

```
import bcrypt
```

Storage process.
1. The user enters the user name and password in the registration form.
2. After receiving this data, the application converts the password into a string of bytes.
3. Use the bcrypt.hashpw function and the automatically generated salt to encrypt the password.
4. If password strength is sage, store the encrypted password in the users table of the MySQL database.

In the code we implement below, we can see two main functions bcrypt.gensalt() and bcrypt.hashpw(). They can generate a new hash.

```
@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.json['username']
        #password = request.json['password']
        password = request.json['password'].encode('utf-8')  # Update: Encode the password to bytes
        recoverykey = request.json['recoverykey']
        # Update: Generate a salt and hash the password
        hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
        cur = mysql.connection.cursor()
        cur.execute("SELECT user_id FROM users WHERE username=%s", (username,))
        account = cur.fetchone()
        if not account:
            if len(password) < 8:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password should be longer than 8 letters'}), 200
            if check_password(password):
                b32_password = base64.b32encode(password.encode()).decode()
                qrcode = pyotp.totp.TOTP(b32_password).provisioning_uri(name=username, issuer_name='WebApp')
                cur = mysql.connection.cursor()
                # cur.execute("INSERT INTO users (username, password,recoverykey,ratelimit) VALUES (%s, %s, %s, %s)", (username, pas
                cur.execute("INSERT INTO users (username, password, secretkey, recoverykey) VALUES (%s, %s, %s, %s)",
                            (username, hashed_password, pyotp.random_base32(), recoverykey))
                mysql.connection.commit()
                return jsonify({'status': 'success', 'message': 'Register Successed', "data": qrcode}), 200
            else:
                return jsonify({'status': 'error', 'message': 'Register Failed: The password has been leaked'}), 200
        else:
            return jsonify({'status': 'error', 'message': 'User has registered'}), 200
    return jsonify({'status': 'success', 'message': 'Register Successed'}), 200
```

Password authentication
1. When users log in, the program will verify that the password they supplied whether matches the encrypted password stored in the database.
2. The program retrieves the associated encryption password by executing a database query using the username.
3. Create a byte string from the password that was input.
4. To compare the password that was input with the encrypted password stored in the database, use the bcrypt.checkpw() function.
5. Then verify in turn whether the user is locked, whether the user name is valid, whether the verification code is correct, whether the password is correct, and possible secondary verification.
6. If all checks pass, the user's login information will be recorded in the session and "success" will be returned to indicate successful login. If any validation fails, an appropriate error message will be returned.

```python
def check_login(userDetails):
    username = userDetails['username']
    # password = userDetails['password']
    password = userDetails['password'].encode('utf-8')  # Encode the password to bytes
    secondpassword = userDetails['secondpassword']
    method = userDetails['method']
    captcha = userDetails['captcha']
    cur = mysql.connection.cursor()
    cur.execute("SELECT user_id,password,limitTime,ratelimit,recoverykey FROM users WHERE username=%s",
                (username,))
    account = cur.fetchone()
    if account[3] >= 5:
        lockuser(username)
        return 'User is been locked'
    if account is None:
        return 'Invalid Username'
    if captcha.lower() != request.cookies.get('captcha').lower():
        error_password(username)
        return 'Invalid Captcha'
    if account[2] is not None and account[2] > datetime.datetime.now():
        return 'User is been locked'
    if account[1] != password:
        error_password(username)
        return 'Invalid Password'
    if not bcrypt.checkpw(password, account[1].encode('utf-8')): # Use bcrypt to check passwords
        error_password(username)
        return 'Invalid Password'
    if method == '0':
        # b32_password = base64.b32encode(password.encode()).decode()
        b32_password = base64.b32encode(password).decode()
        totp = pyotp.TOTP(b32_password)
        result = totp.verify(secondpassword)
        if not result:
            error_password(username)
            return 'Invalid Second Password'
    else:
        if account[4] != secondpassword:
            error_password(username)
            return 'Invalid Second Password'
    session['username'] = username
    session['user_id'] = account[0]
    return 'success'
```

5.2 Utilized Libraries and Tools

1. Flask: This framework can be used to configure routing to process and produce responses for POST-type http requests that are sent by the front end. Here's an illustration.

```python
@app.route( rule: '/send_message', methods=['POST'])
def send_message():
    if not request.json or not 'ciphertext' in request.json or not 'iv' 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)

    sender_id = session['user_id']
    receiver_id = request.json['receiver_id']
    message_text = request.json['ciphertext']
    iv = request.json['iv']
    tag = request.json['tag']

    save_message(sender_id, receiver_id, message_text, iv, tag)

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

This code first configures a route to handle post requests, which are made in response to user queries for the URL "/send_message." Requests from users are handled in the second section. A request format error will be returned if the user's request does not meet the requirements. After the request format is verified, the response is produced using the 'jsonify()' function.

2. Flask-Session: Using Flask-Session, we may save session data like ‘user_id’ on the server prevent tampering.

```python
# Configure secret key and Flask-Session
app.config['SECRET_KEY'] = 'your_secret_key_here'
app.config['SESSION_TYPE'] = 'filesystem'  # Options: 'filesystem', 'redis', 'memcached', etc.
app.config['SESSION_PERMANENT'] = False
app.config['SESSION_USE_SIGNER'] = True  # To sign session cookies for extra security
app.config['SESSION_FILE_DIR'] = './sessions'  # Needed if using filesystem type
```

This library makes session persistence secure. SESSION_PERMANENT and SESSION_USE_SIGNER parameters aid in the protection of session data from tampering.

3. Flask-Mysqldb: Flask-Mysqldb allows our program to connect to and manage the MySQL database. With the help of this library, we can manage some important data such as, user passwords, chat secret information and so on by running specific SQL statements.

```python
    cur = mysql.connection.cursor()
    query = """SELECT message_id,sender_id,receiver_id,message_text,iv,tag,second_tag 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))
```

4. PyYAML: Code and configuration data are separated by this library. Without changing the code, we can alter the database connection by changing the contents of the db.yaml file.

```
# Load database configuration from db.yaml or configure directly here
db_config = yaml.load(open('db.yaml'), Loader=yaml.FullLoader)
app.config['MYSQL_HOST'] = db_config['mysql_host']
app.config['MYSQL_USER'] = db_config['mysql_user']
app.config['MYSQL_PASSWORD'] = db_config['mysql_password']
app.config['MYSQL_DB'] = db_config['mysql_db']

mysql = MySQL(app)
```

Derivation and Storage of Key Materials

how key materials are derived：

```
async function initialization() {
    let ecKeys = await crypto.subtle.generateKey({name: "ECDH", namedCurve: "P-384"}, true, ["deriveKey"],);
    let jwk_PublicKey = await crypto.subtle.exportKey("jwk", ecKeys.publicKey);
    let jwk_PrivateKey = await crypto.subtle.exportKey("jwk", ecKeys.privateKey);

    const response = await fetch('/push_public_key', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({posterID:myID, publicKey: jwk_PublicKey}),
    });
```

1） Create an ECDH key pair (a public key and a private key) first using the Web Crypto API. Subsequently, encode the public key using JWK format and transmit it to the server using a post-type http request.

```
@app.route( rule: '/push_public_key', methods=['POST'])
def register_public_key():
    data = request.json
    posterID = data['posterID']
    publicKey[f"User{posterID}PK"] = data['publicKey']
    print(f"User{posterID}PK:", publicKey[f"User{posterID}PK"])
    return jsonify({'message': 'Public key registered successfully'}), 200
    # return f"{session}", 200
```

2） The backend provides a separate route for receiving "post" requests with the public key and storing it.

```
try{
    const response = await fetch(`/pull_public_key/${peer_id}`,{
        method: 'GET',
        headers:{
            'Accept':'application/json',
        }
    });

    if (!response.ok){
        const data = await response.json();
        console.error("Error in HTTP:",data.message)
    }

    const { publicKey } = await response.json();
    const peerPublicKey = await crypto.subtle.importKey(
            "jwk",
            publicKey,
            {
                name: "ECDH",
                namedCurve: "P-384"
            },
            true,
            [],
    );
    let jwk_peerPublicKey = await crypto.subtle.exportKey("jwk", peerPublicKey);
```

3) Send an http request of “get” type to request the user's public key from the server. Convert the public key obtained from the server in JSON format into a format that the "crypto" library can operate on. Then it is converted into JSON format again and stored in local storage.

```
@app.route( rule: '/pull_public_key/<peer_id>', methods=['GET'])
def get_public_key(peer_id):
    public_key = publicKey.get(f"User{peer_id}PK")
    print(f"Data retreived for {peer_id}: ", public_key)
    if public_key:
        return jsonify({'publicKey': public_key})
    else:
        return jsonify({'message': 'Public key not found'}), 404
```

4) The back-end code sets the route to obtain the request and provides the corresponding public key.

```
async function deriveKeys(peerPublicKey, salt=Salt){
    try {
        const privateKey = await importKey("privateKey")
        const sharedSecret = await crypto.subtle.deriveKey(   //Deriving shared secret
            {
                name: "ECDH",
                namedCurve: "P-384",
                public: peerPublicKey,
            },
            privateKey,
            {
                name: "HKDF",
                hash: "SHA-256",
                length: 128,
            },
            false,
            ["deriveKey"]
        )
```

Use your own private key and the other party's public key to derive a shared key (using the ECDH algorithm, the magic of mathematics). This shared key will not be transmitted in the network but calculated by each party using their own private key and the other party's public key, ensuring the security of the key.

```
const AESOptions = {
    "algorithm":
        {
            name: "HKDF",
            hash: "SHA-256",
            salt: numToUint8Array(salt),
            info: new Uint8Array(`CHAT_KEY_USER${myID}to${peer_id}`),
        },
    "derivedKeyAlgorithm":
        {
            name: "AES-GCM",
            length: 256,
        },
}

const MACOptions = {
    "algorithm":
        {
            name: "HKDF",
            hash: "SHA-256",
            salt: numToUint8Array(salt),
            info: new Uint8Array(`CHAT_MAC_USER${myID}to${peer_id}`),
        },
    "derivedKeyAlgorithm":
        {
            name: "HMAC",
            hash: "SHA-256",
            length: 256,
        },
}
```

```
const AESKey = await crypto.subtle.deriveKey(
    AESOptions["algorithm"],
    sharedSecret,
    AESOptions["derivedKeyAlgorithm"],
    true,
    ["encrypt", "decrypt"]
)

const MACKey = await crypto.subtle.deriveKey(
    MACOptions["algorithm"],
    sharedSecret,
    MACOptions["derivedKeyAlgorithm"],
    true,
    ["sign", "verify"]
)

return {
    AESKey: AESKey,
    MACKey: MACKey,
}
```

6) Use the derived shared key as a parameter of the HKDF algorithm to derive the AES key and MAC key. (The specific implementation method will be explained later in the content).

How we store key materials and the size of them:

Private keys, own public keys, others‘ keys, and AES keys and MAC keys all exist in local storage.

The server only stores public keys (the front end will request other users' public keys and store them in local storage).

```
localStorage.setItem("publicKey",JSON.stringify(jwk_PublicKey));
localStorage.setItem("privateKey", JSON.stringify(jwk_PrivateKey));
console.log("Public key: ", localStorage.getItem("publicKey"));
console.log("Private key: ", localStorage.getItem("privateKey"));
```

When the public key and private key are created, they will be stored directly in local storage in JSON format.

```
const response = await fetch('/push_public_key', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({posterID:myID, publicKey: jwk_PublicKey}),
});
```

The user's public key is sent to the backend via a "post" type request.

```
@app.route( rule: '/push_public_key', methods=['POST'])
def register_public_key():
    data = request.json
    posterID = data['posterID']
    publicKey[f"User{posterID}PK"] = data['publicKey']
    print(f"User{posterID}PK:", publicKey[f"User{posterID}PK"])
    return jsonify({'message': 'Public key registered successfully'}), 200
```

The backend has a dedicated route to receive requests and save the public key in the dictionary.

```
const peerPublicKey = await crypto.subtle.importKey(
        "jwk",
        publicKey,
        {
            name: "ECDH",
            namedCurve: "P-384"
        },
        true,
        [],
);
let jwk_peerPublicKey = await crypto.subtle.exportKey("jwk", peerPublicKey);
localStorage.setItem(`peerPK_${peer_id}`, JSON.stringify(jwk_peerPublicKey));
```

After obtaining other users' public keys as described previously, Save it in JSON format.

```
if (AES_MAC){
    crypto.subtle.exportKey("jwk", AES_MAC.AESKey).then(jwkAES => {
        localStorage.setItem("AESKey", JSON.stringify(jwkAES))
        console.log("AES Key derived successfully: ", AES_MAC.AESKey);
    })

    crypto.subtle.exportKey("jwk", AES_MAC.MACKey).then(jwkMAC => {
        localStorage.setItem("MACKey", JSON.stringify(jwkMAC))
        console.log("MAC Key derived successfully: ", AES_MAC.MACKey);
```

Convert the generated AES key and MAC key from 'CryptoKey' format to JWK format, then use "stringify()" to convert them into JSON strings and store them in local storage.

For public and private keys: Since it is specified that P-384 elliptic curve must be used when generating keys, the length of all public and private keys should be changed to 384 bits.

```
let ecKeys = await crypto.subtle.generateKey({name: "ECDH", namedCurve: "P-384"}, true, ["deriveKey"],);
let jwk_PublicKey = await crypto.subtle.exportKey("jwk", ecKeys.publicKey);
let jwk_PrivateKey = await crypto.subtle.exportKey("jwk", ecKeys.privateKey);
```

For AES key and MAC key: the length is 256 bits.

```
const AESOptions = {
    "algorithm":
        {
            name: "HKDF",
            hash: "SHA-256",
            salt: numToUint8Array(salt),
            info: new Uint8Array(`CHAT_KEY_USER${myID}to${peer_id}`),
        },
    "derivedKeyAlgorithm":
        {
            name: "AES-GCM",
            length: 256,
        },
}

const MACOptions = {
    "algorithm":
        {
            name: "HKDF",
            hash: "SHA-256",
            salt: numToUint8Array(salt),
            info: new Uint8Array(`CHAT_MAC_USER${myID}to${peer_id}`),
        },
    "derivedKeyAlgorithm":
        {
            name: "HMAC",
            hash: "SHA-256",
            length: 256,
        },
```

5.4 Generation of Domain Certificate

```
openssl ecparam -out webkey.pem -name secp384r1 -genkey
```

We firstly create the ‘webkey’.

```
openssl req -new -out webcsr.csr -key webkey.pem -config openssl.cnf
```

Secondly, we generate the certificate request.
The picture of ‘openssl.cnf’ file is as follows:

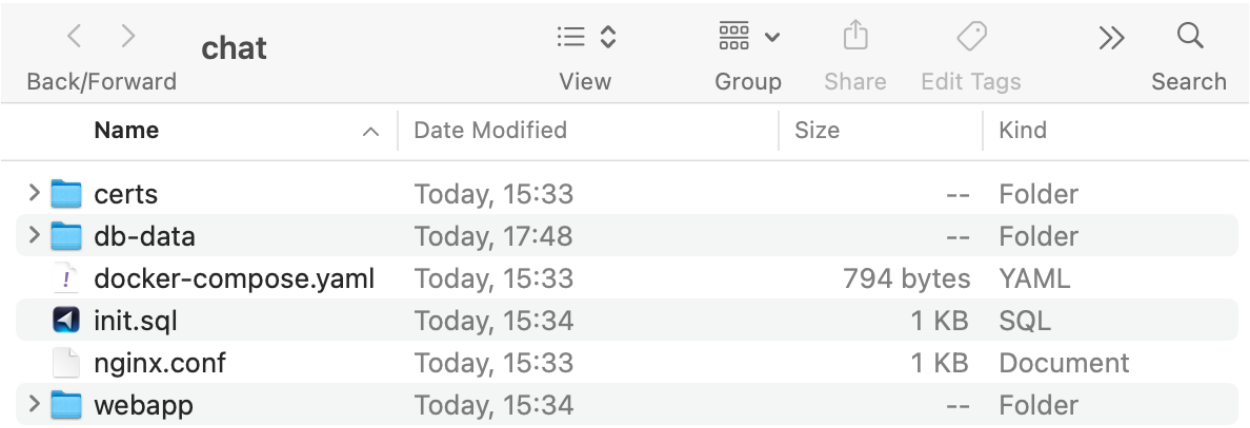
```
[ req ]
default_bits = 384
prompt = no
default_md = SHA384
distinguished_name = dn
x509_extensions = v3_req

[ dn ]
countryName         = HK
commonName          = group-31.comp3334.xavier2dc.fr

[ v3_req ]
subjectAltName = @alt_names
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:false
keyUsage = critical,digitalSignature
extendedKeyUsage=serverAuth

[ alt_names ]
DNS.1 = group-31.comp3334.xavier2dc.fr
```

```
openssl x509 -req -extfile openssl.cnf -extensions v3_req -in webcsr.csr -out web.crt -CA cacert.crt -CAkey cakey.pem -CAcreateserial -days
```

Lastly, we generate the certificate.

6. System Deployment

1. Download the whole project and "cd" the current working path to the path of "chat" which include docker-compose.yaml



2. Deploy docker container using the command: docker-compose up -d.

```
chenziyang@chenziyangs-MacBook-Air ~ % cd /Users/chenziyang/Desktop/chat
chenziyang@chenziyangs-MacBook-Air chat % docker-compose up -d
[+] Running 3/4
 ⠿ Network chat_default      Created                                    0.4s
 ✔ Container chat-db-1       Started                                    0.2s
 ✔ Container chat-webapp-1   Started                                    0.2s
 ✔ Container chat-nginx-1    Started                                    0.3s
```

3. The chat app works over plain HTTP on port 8443, and the website is deployed include our group number “31”, and you can open Chrome or Firefox to access it at: https://group-31.comp3334.xavier2dc.fr:8443
4. Open a new private window of your chosen browser and access the website https://group-31.comp3334.xavier2dc.fr:8443 again.