COMP3334 Group Project Report

End-to-end encrypted chat web application

LIU Chenxi (21096794D), LIU Chengju (21100052D), DU Haoxun (21097883D),
LIANG Zhihong (21094107D), WANG Yukai (21094692d)

1. Introduction

1.1 Background:

This message “age of browser providers occupy the topmost place in customer interactions on digital services” narrates unmatched and unchallenging position of the web browser as all access portal to the services. The watershed moment when our interactions with interfaces surpass physical interactions at work and extracurricular lives, the question of online security for web communications turns from one of privacy to one of the right or the manners of internet usage. Network security has been shaped by the element of privacy to becoming one of the critical components as a result of the increased use of modern technological devices.

This report aims to present a task based on turning a straightforward chat app into an E2EE (end-to-end encryption)-an equipped web app that aims to ensure that all communication between the end users is encrypted. The task is fueled by a set of multi-faced objectives that serve as building blocks of the comprehensive security framework. The endeavor is guided by a multifaceted set of objectives, each contributing to a comprehensive security framework. A multifaceted set of objectives guides the endeavor, each contributing to a comprehensive security framework:

a) **Adaptation to Secure E2EE**: Converting a standard chat program into an encrypted application to avoid being breached by any third party and, thus, the messages being inaccessible to the intended receiver other than the sender.
b) **Compliance with NIST SP 800-63B**: The rigorous standards set in the report and guidelines published by the National Institute of Standard and Technology (NIST), Special Publication 800-63B Authentication and Lifecycle Management – Digital Identity Guidelines, explicitly apply to federated agencies (US) and serve as the basis for different systems.
c) **MFA Implementation**: Implementation of a robust MFA (Multi-Factor Authentication) involving the merger of passwords with either OTP (One-Time Passwords) or mechanisms developed under the new FIDO2 standard so that a well-labeled line of defense against undesired access is created.
d) **E2E Encryption**: Implementation of end-to-end encryption algorithms that don't allow the server that links the users to decode the actual content between them.
e) **Securing Communications in Transit**: The set-up of recent TLS, a transport mechanism that ensures data safety during transmission via the network, is carried out to prohibit eavesdropping and man in the middle attacks.
f) **Dockerization**: The packaged Docker container image, offering scaling, transportation, and deployment advantages, regardless of the environment it is in, will eliminate the need for compromises regarding security.

In the second part of this study, we will focus on the strategy itself that has been adopted to reach those objectives. Technical interventions, security considerations, and the strategy design will be explained in detail. The project will be a secure

communication platform. Compliance with such regulations includes assessing and applying recognized standards in full measure. As a result, the project guarantees future web application development.

1.2 Environment Setting & Packages

The environment setting and packages used are shown in Table 1

Package	Version
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
argon2-cffi	23.1.0
pyotp	2.9.0
Flask-limiter	3.5.1

Table 1 Environment Setting

The details of the packages could be figured out through the document "requirements.txt."

1.3 Duty Division

The project's duties are divided into authentication, E2EE Chat, and TLS. We have 5 individuals in the group, and we divide "2-2-1" for each component, as indicated in Table 2.

Member	Part
LIANG Zhihong	Authentication
DU Haoxun	Authentication
LIU Chenxi	E2EE Chat
LIU Chengju	E2EE Chat
WANG Yukai	TLS

Table 2 Group Duties Division

2. Implemented Features and Requirements

The implemented features and requirements are shown in Table 3 as follows.

Requirement	Origin	Implement
User-chosen Memorized Secret	Authentication	Done
Single-Factor OTP Device (Google Authenticator)	Authentication	Done
Rate-limiting Mechanisms	Authentication	Done

Image-based CAPTCHAs	Authentication	Done
New Account Registration and Bind Authenticators	Authentication	Done
Memorized Secret Verifiers	Authentication	Done
ECDH Key Exchange	E2EE Chat	Done
Key Derivation	E2EE Chat	Done
Encrypt messages with AES in GCM mode	E2EE Chat	Done
Key Storage	E2EE Chat	Done
Message History	E2EE Chat	Done
Key Refresh	E2EE Chat	Done
Key Exchange Initiation	E2EE Chat	Done
Encode chat messages and Format network message	E2EE Chat	Done
Logging	E2EE Chat	Done
Security Protections	E2EE Chat	Done
TLS Configuration	TLS	Done
TLS Certificate Specs	TLS	Done
Website Hosting	TLS	Done
Hosts File Adjustment	TLS	Done
Certificate Issuance	TLS	Done
CA Certificate Constraints	TLS	Done

Table3 Implemented Features and Requirements List

Upon thoroughly examining the project documentation, we have ascertained the requirements and taken appropriate steps to meet each criterion comprehensively. This document presents a detailed exposition of how these requirements have been satisfied in the subsequent section, Section 3.

3. Solution Details

3.1 Authentication

3.1.1 User-chosen Memorized Secret

The User-chosen Memorized Secret refers to a password mechanism where users select their own passwords. According to the NIST guidelines, it should be at least 8 characters and not contain apparent data such as name, birthday, or other easily guessable information.

We adopted the **Argon2** hash algorithm, widely recognized for its strong defense against brute force attacks. This algorithm can perform memory-intensive operations, effectively preventing large-scale automatic guessing attempts and enhancing our selection.

To ensure consistency in password processing and storage methods, we use

Unicode's NFKC normalization technology to normalize user input. This step is essential as it can address the instability in user input, ensuring that when users log in with passwords in different formats (essentially the same), they can still match the stored initial hash value. After normalization, passwords will be hashed using the **'argon2-cffi'** library, which is specifically designed for the secure hashing of passwords. We can configure parameters such as memory cost and hash time using this library to achieve the best balance between performance and security.

Each password hash value is stored with its unique salt and related parameters, such as hash length and computational cost. These measures enhance the security of storing passwords, enabling them to resist rainbow table attacks and brute force attempts. Adding salt ensures that even the same password will generate different hash values. At the same time, parameter storage can cope with future computing power improvements, as parameters can be adjusted to obtain new hash values without affecting existing ones.

3.1.2 Single-Factor OTP Device (Google Authenticator)

This is an OTP authenticator with a time-syncing feature. It provides a time-based OTP which the user must know, making it a double authentication process and enhancing the system security.

We implemented a time-based one-time password (TOTP) algorithm using the 'pyotp' library. This algorithm is particularly effective as it generates a new 6-digit (we even enhanced it to an 8-digit) password every 30 seconds, which the user must enter during the login process. The security of this setup is facilitated by the secure generation and storage of the key, which is unique to each user and is used to generate the seed for the OTP. The secret key is stored in an encrypted format in the user record of the database to prevent unauthorized access.

During the authentication process, the OTP provided by the user is verified against the OTP generated by the server based on the stored secret key computation. This ensures that access is only allowed when both the password and the OTP match, providing a solid defense against unauthorized access. For convenience and added security, the secret key is usually transmitted to the user's device by scanning a QR code during initial setup.

3.1.3 Rate-limiting Mechanisms

Authentication prevention mechanisms such as rate-limiting tend to avoid automatically generated attempts, namely brute force or credential stuffing, by limiting the number of identity verification attempts per IP address in a given timeframe.

To implement this mechanism, we found a certain package under the ‘flask’

which is mainly focus on the rate-limiting issue – 'flask_limiter'. While 'flask_limiter' could help us with the prohibition, we have another method to get the IP from the user which is 'get_remote_address' to figure out the IP who is sending the requests, banning will also focus on certain IP. The initialization of the limiter is shown in Figure4.1.

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

```python
@app.route('/login', methods=['GET', 'POST'])
@limiter.limit('100 per day')
```

```python
@app.errorhandler(429)
def ratelimit_handler(e):
    return render_template('rate_limit_exceeded.html'), 429
```

Figure4.1 Configuration of the limiter

The default value of the limiter is set randomly, while the details will be stated during the login route which is also shown in the Figure4.1, and the final limitation is set to meet the standard 100 requests per day.

To improve the user experience, we also modify the default prompt website to make the user fully aware of the current state if they have been banned as a result of the exceed the limitation with adding a function 'ratelimit_handler' to automatically catch the error if occurs from the using process (the implementation is shown in the Figure4.1). The page will be transmitted to is shown in the Figure4.2.

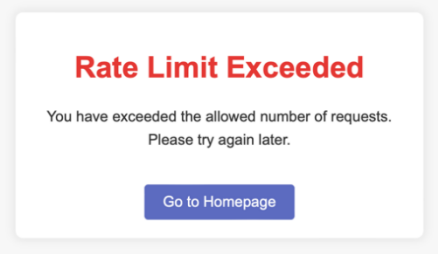


Figure4.2 Rate-limiting Prompt HTML

3.1.4 CAPTCHAs

This feature plays a key role in differentiating the Human users from the Bots in the login process. The input code typed by users could be used for Captcha and can be compared with the generated part of authentication.

At first, we meet trouble when implementing as assuming that the function is based on python code. But after trying, we finally understand that it is expected to build on the html and JavaScript, which makes us understand more about the website building and CAPTCHAs implementations themselves.

Text-based CAPTCHA

As the function is to check whether the user is human or computer, we make this implementation within the login part as it will become one of the required inputs during the login part. The update on login page is shown in Figure5.1.

Figure5.1 CAPTCHA implementation in Login

Get into the details of the code, the implementation could be explained in two steps:

Step 1 Function initialization in JavaScript

```
function createCaptcha() {
  //clear the contents of captcha div first
  document.getElementById('captcha').innerHTML = "";
  var charsArray =
  "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ@!#$%^&*";
  var lengthOtp = 6;
  var captcha = [];
  for (var i = 0; i < lengthOtp; i++) {
    //below code will not allow Repetition of Characters
    var index = Math.floor(Math.random() * charsArray.length + 1); //get the next character from the array
    if (captcha.indexOf(charsArray[index]) == -1)
      captcha.push(charsArray[index]);
    else i--;
  }
  var canv = document.createElement("canvas");
  canv.id = "captcha";
  canv.width = 100;
  canv.height = 50;
  var ctx = canv.getContext("2d");
  ctx.font = "25px Georgia";
  ctx.strokeText(captcha.join(""), 0, 30);
  //storing captcha so that can validate you can save it somewhere else according to your specific requirements
  code = captcha.join("");
  document.getElementById("captcha").appendChild(canv); // adds the canvas to the body element
  document.getElementById("valid").value = code;
}
```

Figure5.2 Implementation of the createCaptcha in JS

As shown in the Figure5.2, the logic of design of the createCaptcha is: Initial Clearing→Character Set→Captcha Length→Random Selection→Canvas Creation→Text Rendering→Storage and Validation.

This function makes the text-based CAPTCHA generation come true, which could be directly used when there is a need for the versifying user. It stands out for offering a high level of customization and solid security features in an easy-to-integrate package.

Step 2 Function import in html

We call the createCaptcha function at the beginning of the login html, while also creating a place of form to hold the input value of the CAPTCHA which will be read by the users (shown in the Figure5.3). As the user click login button, the form including the CAPTCHA will be submitted to the server and compare each of the value with the database to check whether the user is existed and is a human.

```
<div id="captcha"></div>
<input type="text" placeholder="Captcha" id="cpatchaTextBox" name ="captchaTextBox" required >
```

Figure5.3 Text-based CAPTCAH in login HTML

Image-based CAPTCHA

To meet the requirement for the project, having an image-based CAPTCHA, we figure out a proper way to meet this requirement by using the hcaptcha, which perfectly solve the problem and bring the login route and register route into a more integrate state.

The real case usage of the image-based CAPTCHA is shown in the Figure5.4 as follows.

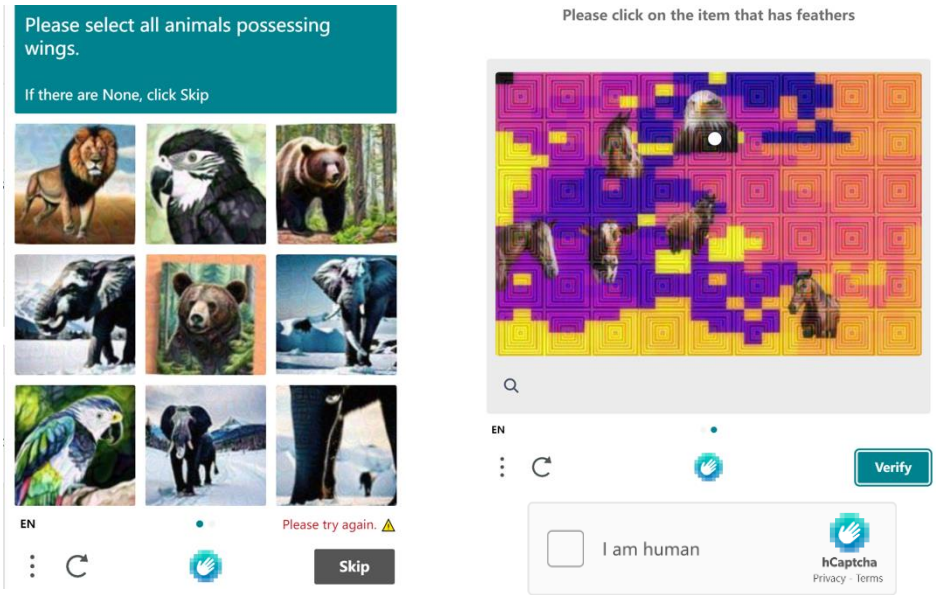


Figure5.4 Image-based CAPTCHA Real Case

As the user click on the column, the website will send request and connect with the hcaptcha website to do the human verification, asking the user to select the right images which follow the description. After user finish the test, the hcaptcha website will return a value back the app which could be figured out the final result of the CAPTCAH test with clear judgment.

3.1.5 New Account Registration and Bind Authenticators

As new user registrations, this can also bind with tools like OTP devices (Google Authenticator) and recovery keys. It serves as an additional level of security because it requires to have multiple authentication factors at the beginning of creating an account.

a) New Account Registration

The registration route follows the step of the login route, which means that the connection building and account verifying is the same as the ones in login route. However, there are several new functions appearing in the registration route.

```python
def check_name_existed(username):
    # Connect to the database
    cur = mysql.connection.cursor()

    # Use sql to check whether there is same username in the database
    cur.execute("SELECT COUNT(*) FROM users WHERE username = %s", (username,))
    search_result = cur.fetchone()[0]
    number_of_existing = int(search_result)

    # Check whether the number of username is more than 0
    if number_of_existing > 0:
        # Already existing
        return True
    else:
        # Not existing
        return False
```

Figure6.1 check_name_existed Function in Registration

check_name_existed function shows the goal to validate whether the new username is already in the database because of the requirement that we do not allow there are repetitive names conflict. The function will clearly check with the database to find whether the new name is valid or not, while the output is Boolean value that may help judgment in the registration route.

Figure6.2 register_new Function in Registration

register_new function shows the implementation that we use SQL command to add a new account record into the database. This function will be called if the new account meets all the requirements which are listed in the registration route, simplifying account is registered successfully.

b) Bind Authentication

After the account meets the registration requirements, the app will immediately encrypt the input password with the Argon2, replacing the origin password with the hashed one (shown in the Figure6.3).

```python
# initialize an argon2 password hasher
ph = argon2.PasswordHasher()

# hash password
password_hash = ph.hash(password)
```

Figure6.3 Argon2 Hashes User Password

To increase the security of the whole system, we add on a TOTP secret with its QR code, which will be automatically generated after

registration and show to the user. The user needs to bind scan the QR code with an authenticator app like Google Authenticator. After the registration, every time users are required to input the OTP. The system will randomly generate a 256 bits TOTP secret, and the secret will be shared to the user with a QR code (shown in the Figure6.4).

```
# generate TOTP secret
totp_secret = pyotp.random_base32()
# generate QRCode url
auth = pyotp.totp.TOTP(totp_secret, digits=8).provisioning_uri(name=username, issuer_name="COMP3334 Group 12")
```

Figure6.4 TOTP Secret and QR code Generate

To solve with the situation when users lost their authenticators, we add a function to add a recovery key to every account when registration which could help the users to regain their account. The information entropy of the recovery key is approximately 45.6 bits.

$$\log_2 52^8 \approx 45.6 \text{ bits}$$

```
# randomly generate recovery keys securely
def generate_recovery_keys():
    keys = []
    for i in range(2): # 6 recovery keys
        key = ''
        for j in range(8): # 8 digits
            key += str(''.join(secrets.choice(string.ascii_letters)))
        keys.append(key)
    return keys

def hash_recovery_key(keys):
    hashed_recovery_keys = []
    ph = argon2.PasswordHasher()
    for k in keys:
        hashed_recovery_keys.append(ph.hash(k))
    return hashed_recovery_keys
```

Figure6.5 Recovery Key Generate and Hashing

As shown in the Figure6.5, we first use generate_recovery_keys function to generate the keys and passes the generated keys into the function hash_recovery_key to make them always in a secure state. After hashing, the final results will return to the registration route, and the database will store the hashed recovery key with the account to deal with the mentioned password-forgotten situation afterwards.

3.1.6 Memorized Secret Verifiers

a) Memorized Secrets Encryption

According to NIST, stored secrets (passwords) can be secured by salt and hash with a suitable key derivation function, preventing attacks that decode password databases using precomputed hashing. This implies that the password storage system shall comprise a unique salt for every

password to be refined in such a way that would only produce irreversible hashes, significantly reducing the effectiveness of rainbow table attacks.

We chose **Argon2** as the hashing algorithm because of its ability to balance memory and CPU usage against parallel and serial attacks. The parameters of the algorithm, including salt, memory, and time cost, are stored along with the hashing results. This approach ensures the integrity of the stored passwords and future-proofs our security measures by allowing our system to adjust the hashing difficulty in response to advances in computing power.

b) Memorized Secrets Suggestion

We implemented a system that actively discourages using cracked passwords to complement our encryption efforts. The new model is set up not to instruct the user to select a previously exploited password and then compare it to a previously public password dataset." The **"Have I Been Pwned" API** provides a service called **"Pwned Passwords"** for receiving passwords known to belong to databases whose passwords have been hacked, and we can check in real-time whether a selected password has been previously exposed in a data breach. This proactive measure is critical to preventing the re-use of compromised credentials and significantly strengthens our system's defenses against credential stuffing attacks.

When a user attempts to create or update a password, our system immediately verifies its integrity against a database of known password vulnerabilities. If a password vulnerability is found in this database, the system issues an alert and instructs the user to create a stronger, more secure password. This not only helps maintain the security of user accounts but also teaches users best practices for password security and fosters a more security-conscious user base.

These methods of encrypting memorized secrets and preventing the use of compromised passwords create a robust security framework. This framework not only protects against the immediate threats of unauthorized access but also makes users adhere to best practices in password security, thereby enhancing the overall security posture of our system. Through these measures, we demonstrate a commitment to maintaining and advancing user security in response to evolving cyber threats.

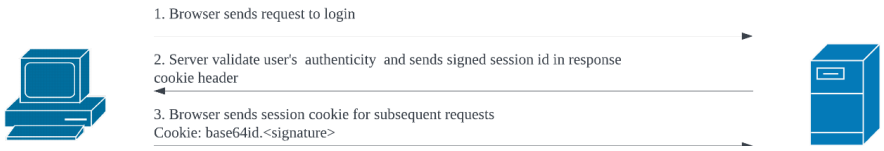
Additionally, we also implemented a password strength meter using **zxcvbn**, a low-budget password strength estimator. It takes in a user's password and returns a score. The score indicates the strength of the

chosen password, and the maximum score is 4. We require the users to have a password with a score higher than 2.

3.1.7 Session Management and Cookie

In this application, we implemented a server-side session. We used the Flask-session library, and the session type is filesystem. With a server-side session, sensitive information can be stored in the server. Each session will be assigned a session id. The session id is generated using **secrets.token_urlsafe** which is a random base64 encoded text string. Also, the server has a secret key generated by **secrets.token_hex**. It is a 32 bytes random string in hexadecimal. Then the session id is signed by the server secret key. The value of the session cookie is the session id concatenated with the signature. After user successful user authentication, the session cookie is sent as a response to the browser.

The session cookie is not permanent which means that it will be deleted when the browser is closed. Moreover, the cookie is secure and httponly. The same site attribute is set to Strict to prevent CSRF attacks.



3.1.8 Prevention of XSS, CSRF and SQL injection

XSS: XSS protection is internally implemented in Flask. It configures Jinja2 to automatically escape all values. One thing that we need to protect is XSS by attribute injection. We counter this factor by quoting every attribute when using Jinja expressions in them.

```
<p>Don't have an account? <a href="{{ url_for("register") }}">Register</a> </p>
```

CSRF: We used Flask-WTF to implement CSRF protection. In every html form and HTTP request header, we included a CSRF token. Then, the when the server are handling requests, it will check the CSRF token. The server will only process the request with a valid CSRF token

```
<input type="hidden" name="csrf_token" value = "{{ csrf_token() }}" />
```

SQL: To counter SQL injection attack, we made sure to use the named attributes when we are passing the query parameters. Then the database will use the specific type and value of the attribute to execute the query.

```
cur.execute("SELECT COUNT(*) FROM users WHERE username = %s", (username,))
```

Before we insert data into the database, we input the invalidation of the message. We also checked whether the data structure is consistent with the

structure we defined at the beginning of the chapter. Also, check whether the value makes sense (e.g., whether this is a key or not or whether the message is a buffer array string).

3.2 E2EE CHAT

The E2EE chat app implements rigorous security practices that provide sufficient security for users to engage in a private and secure conversation, having peace of mind about the strength of the underlying cryptographic protocols and implementations.

Data structure for JSON network message:

```
const payload = {
    receiver_id: peer_id,
    message_text: message,
    message_type: type,
    message_iv: iv,
    message_value: value,
    message_tag: tag,
};
```

message_type	message_text	message_iv	message_value	message_tag	receiver_id
normal	ciphertext	iv used	''	mac tag	peer_id
ECDH request/ECDH response	public key	''	''	''	peer_id
refresh key	''	the last iv	old mac tag	new mac tag	peer_id
erase chat	''	''	''	''	peer_id

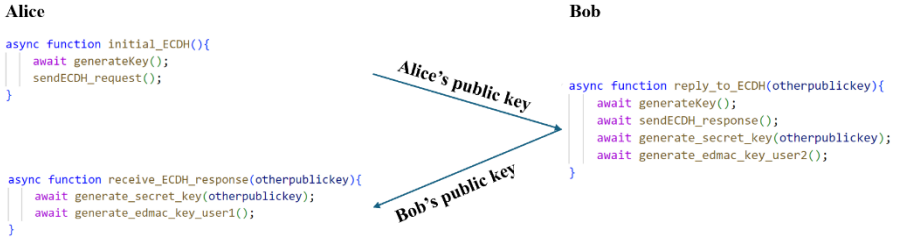
Data structure for local storage:

This overviews the variables stored in local storage for the user (j). The details of the initialization and usage of the variables will be introduced in more information in the following session.

Variable name	Description/usage
USER(i)	Checking flag. If this variable exists in the local storage, the key materials with user (i) exist.
valid_since_USER(i)	The created time of the oldest message that can be decrypted with user(i)
last_response_USER(i)	The created time of the last responded message with user (i)
secret_START_USER(i)	The secret counter used to decrypt the oldest message that can be decrypted with user (i)
keyset_START_USER(i)	The keyset counter used to decrypt the oldest message that can be decrypted with user (i)
decryption_iv_START_USER(i)to(j)	The decryption iv used to decrypt the oldest message that can be decrypted from user(i) to (j)
decryption_iv_START_USER(j)to(i)	The decryption iv used to decrypt the oldest message that can be decrypted from user(j) to (i)
secret_counter_USER(i)	The secret bit counter indicates which secret we are using now with user (i)
secretbits_USER(i)_COUNTER(s)	The s-th secret bits used with user(i)
keyset_counter_USER(i)_SECRET(s)	The keyset counter with user (i) under secret(s)
CHAT_KEY_USER(i)to(j)_KEYSET(c)_SECRET(s)	The encryption/decryption key for message send from user(i) to user(j) using keyset(c) under secret(s).
CHAT_KEY_USER(j)to(i)_KEYSET(c)_SECRET(s)	The encryption/decryption key for message send from user(k) to user(i) using keyset(c) under secret(s).
encryption_iv_USER(i)_KEYSET(c)_SECRET(s)	The number of messages encrypted using keyset(c) under secret(s) with user(i)
decryption_iv_USER(i)to(j)_KEYSET(c)_SECRET(s)	The number of messages encrypted from user(i) to user(j) using keyset(c) under secret(s)
decryption_iv_USER(j)to(i)_KEYSET(c)_SECRET(s)	The number of messages encrypted from user(j) to user(i) using keyset(c) under secret(s)
CHAT_MAC_USER(i)to(j)_KEYSET(c)_SECRET(s)	The MAC key to sign the message from user(i) to user(j) using keyset(c) under secret(s)
CHAT_MAC_USER(j)to(i)_KEYSET(c)_SECRET(s)	The MAC key to sign the message from user(j) to user(i) using keyset(c) under secret(s)

3.2.1 ECDH Key Exchange

Workflow: Use the ECDH key exchange protocol to establish a shared secret between two users.



Step1. Generate my secret and public key pair:

From the webcrypto API, we use the function **generateKey()** to generate the secret-public crypto key pair. We chose P-384 as the underlying curve as

required. We also use **exportKey()** to make the public key available in an external, portable format, **"jwk."** Send the crypto key in jwk format to the peer.

```
async function generateKey() {
  myKeyPair = await window.crypto.subtle.generateKey(
    {
      name: "ECDH",
      namedCurve: "P-384",
    },
    true,
    ["deriveBits","deriveBits"],
  );
  myExportPublicKey = await window.crypto.subtle.exportKey("jwk",myKeyPair.publicKey);
  console.log(myExportPublicKey);
}
```

Output log for the exported public key.

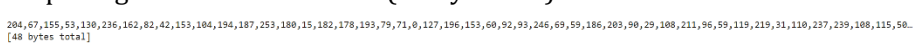
Step2. Generate shared secret bits after receiving other's public key:
Remember that we export the key before sending the key. So, as a receiver, we need to import the key using the same type, **"jwk."**

```
async function generate_secret_key(otherPublicKey){
  const publicKey = await window.crypto.subtle.importKey(
    "jwk",
    JSON.parse(otherPublicKey.replaceAll("\'", "\"").replace("True","true")),
    {
      name: "ECDH",
      namedCurve: "P-384",
    },
    true,
    []
  );
  await deriveSharedSecret(
    myKeyPair.privateKey,
    publicKey
  );
}
```

Then, we use **deriveBits()** to get the shared secret. We didn't derive the shared secret key because we found that it cannot be extractable, which means it can't be exported and stored. So, we derive bits first and store them in local storage. In this case, we can reuse the shared secret to generate different sets of key pairs.

```
async function deriveSharedSecret(privateKey, publicKey) {
  const secret = await window.crypto.subtle.deriveBits(
    { name: "ECDH", public: publicKey},
    privateKey,
    384
  );
  const buffer = new Uint8Array(secret, 0, 48);
  content = `${buffer}…[${secret.byteLength} bytes total]`;
  console.log(content)
}
```

Output log for the shared secret (Array Buffer).



Message exchange:

As stated in the network JSON format message data structure, the following shows the message log.


```
{"receiver_id":2,"message_text":{"crv":"P-384","ext":true,"key_ops":[],"kty":"EC","x":"gxR-
TOlQ5C0dm634t2DetaSBUfWX3bROzBJUpnoIPs4agC2K761eEG09ePxaCQi_","y":"QhlV_LjJ56aFlw9wdAvjfITIPVbE2P_WSXha7krONcZz006wnizXGFKq7Wc58hTh"},"message_type":"ECDH
request","message_iv":"","message_value":"","message_tag":""}

{"receiver_id":1,"message_text":{"crv":"P-384","ext":true,"key_ops":[],"kty":"EC","x":"KJLs4GRzFupFnaHh1_PEizE-E9Qx7piZtvw4RwmPZrbsASq3zpAD_2RaRJBCSK7x","y":"ZVi-CPA-
HKf6Plh6QeFE7EwZ9psAj7FR-j7K3yPYx_KGWqZw-E9kLU1eg3_OKkNw"},"message_type":"ECDH response","message_iv":"","message_value":"","message_tag":""}
```


Local storage:

Key	Value
keyset_counter_USER2_SECRET1	0
secret_counter_USER2	1
secretbits_USER2_COUNTER1	204,67,155,53,130,236,162,82,42,153,104,194,187,253,180,15,182,178,193,79,71,0,127,196,153,60,92,93,246,69,59,186,203,90,29,108,211,...

We store the secret bits in the string format for an array buffer. We also store the secrets we use now. This is represented by **'secret_counter_USER2'**. Each user can have different secrets because the other peer would clean the local storage and start several ECDH requests. The **'keyset_counter_USER2_SECRET1'** represents the counter for the keyset we should use derived from secret key 1. Since we just finished the derived secret and haven't started the derived keys, the counter is 0.

3.2.2 Key Derivation

To utilize the key generated from the ECDH exchange, we derive two 256-bit AES-GCM encryption keys and two 256-bit MAC keys from the shared secret using HKDF-SHA256.

Workflow:

Step 1. Get secret key

First, we check the info stored in local storage and select the corresponding shared secret bits. Then, we import the shared secret key from shared secret bits. Now we get the shared secret key.

```
function get_shared_secret_key(){
  let counter = JSON.parse(localStorage.getItem("secret_counter_USER"+peer_id.toString()));
  let secretfromstorage = parseArrayBufferString(localStorage.getItem("secretbits_USER"+peer_id.toString()+"_COUNTER"+counter.toString()));

  return window.crypto.subtle.importKey(
    "raw",
    secretfromstorage,
    { name: "HKDF" },
    false,
    ["deriveKey"]
  );
}
```

Step 2. Update keyset counter

We must increase the keyset counter since we are deriving new keys from this secret.

Step 3. Derive encryption/decryption key

Derive two 256-bit AES-GCM encryption keys using HKDF-SHA256.

```
function getKey(keyMaterial, saltnum, information) {
  return window.crypto.subtle.deriveKey(
    {
      name: "HKDF",
      salt: saltnumToUint8Array(saltnum),
      info: new Uint8Array(information),
      hash: "SHA-256",
    },
    keyMaterial,
    { name: "AES-GCM", length: 256 },
    true,
    ["encrypt", "decrypt"]
  );
}
```

The salt we used is precisely related to the keyset counter. For each derivation for the same shared secret, the counter will increase. So, the salt equal to the counter will not be repeated. The information represents the current context (e.g., "CHAT_KEY_USER1to2").

Step 4. Export and store encryption/decryption key

We export keys as a **'raw'** type and store the array buffer in the local storage. When storing keys, we need to combine the keyset and secret bits counter with the keys.

Step 5. iv initialization

We store the iv used for encrypting messages, iv used for decrypting messages from others, and iv used for decrypting messages from myself. The IV storage is needed because we want to keep track of a number of messages encrypted using the key pair and a number of message decryptions for further IV invalidation and older message decryption. Since all the keys are not used yet, the iv values are all 0.

Step 6. Derive mac keys

Derive two 256-bit MAC keys from the shared secret using HKDF-SHA256.

```
function getmacKey(keyMaterial, saltnum, information) {
  return window.crypto.subtle.deriveKey(
    {
      name: "HKDF",
      salt: saltnumToUint8Array(saltnum),
      info: new Uint8Array(information),
      hash: "SHA-256",
    },
    keyMaterial,
    { name: "HMAC",
      hash: "SHA-256",
      length: 256 },
    true,
    ["sign", "verify"]
  );
}
```

The salt is related to the keyset counter. So, the salt will not repeat.
The information represents the current context (e.g., "CHAT_MAC_USER1to2").

Step 7. Export and store MAC keys.
We export the key using the **'jwk'** format. And store it with a keyset counter and secret bits counter.

Local storage:

secretbits_USER2_COUNTER1	246,57,92,75,149,1,93,10,125,206,73,2,72,127,209,123,66,145,141,246,40,251,107,192,5,37,188,92,24,176,42,120,49,245,14...
keyset_counter_USER2_SECRET1	1
secret_counter_USER2	1
CHAT_KEY_USER1to2_KEYSET1_...	254,187,38,39,136,179,122,144,173,66,121,198,101,111,64,48,19,144,141,173,49,129,180,195,107,232,143,96,191,189,231,26
CHAT_KEY_USER2to1_KEYSET1_...	24,213,51,143,147,151,137,161,30,125,20,221,95,98,248,207,128,75,208,85,108,6,1,209,180,204,15,81,193,115,106,201
encryption_iv_USER2_KEYSET1_...	0
decryption_iv_USER2to1_KEYSE...	0
decryption_iv_USER1to2_KEYSE...	0
CHAT_MAC_USER1to2_KEYSET...	{"alg":"HS256","ext":true,"k":"_rsmJ4izepCtQnnGZW9AMBOQja0xgbTDa-iPYL-95xo","key_ops":["sign","verify"],"kty":"oct"}
CHAT_MAC_USER2to1_KEYSET...	{"alg":"HS256","ext":true,"k":"GNUzj5OXiaEefRTdX2L4z4BL0FVsBgHRtMwPUcFzask","key_ops":["sign","verify"],"kty":"oct"}

3.2.3 Normal message sending and receiving

Sender:
(a) encrypt messages with AES in GCM mode
Messages are secured by the Advanced Encryption Standard (AES) implemented in Galois/Counter Mode (GCM).
Workflow:

First, we encode the message. Then, we get information from local storage. We need the secret and keyset counter to get the encryption key. iv. We then use '**encrypt()**' to get the ciphertext. At last, update the encryption iv. The iv is increased every time a message is encrypted, so it will not be repeated for the same encryption key.
Output log of ciphertext: `cyphertext value 28,191,183,77,13...[22 bytes total]`

(b)Protect the IV with HMAC-SHA256
Protect the IV with HMAC-SHA256 using the derived MAC key to prevent the attacker from choosing IVs.
Workflow

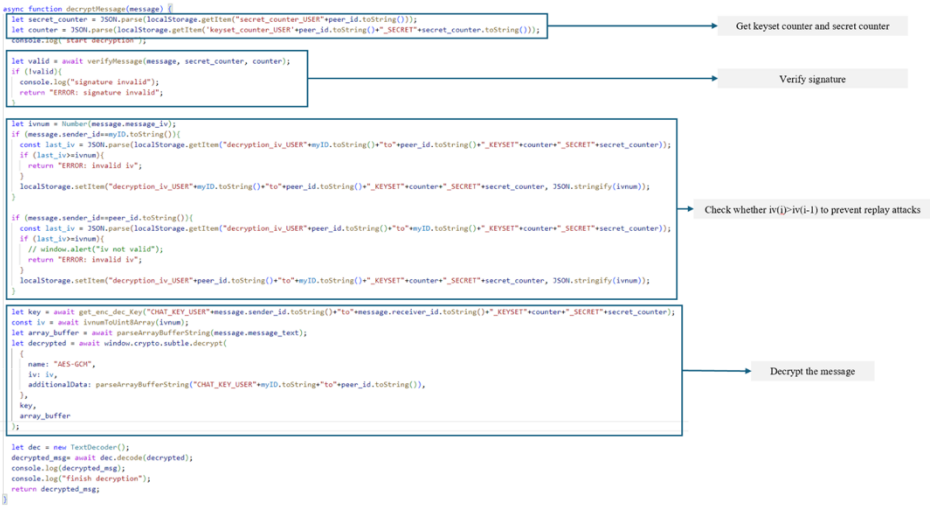
First, get iv and encode it with additional data altogether. Generate the signature using **'sign()'** by my mac key. For normal messages, the additional data is empty.

Output of signature log:

```
signature:
120,181,182,237,54,195,89,27,23,156,178,3,44,178,61,204,145,111,45,215,157,104,40,35,251,122,26,170,187,231,85,160
```

Receiver:

Workflow



Step1. Get keyset counter and secret counter

Step2. Verify signature.

We use the **'verify()'** function with the other's MAC key, signed iv, and signature to verify the signature.

Step3. Check iv.

To avoid a replay attack, we check whether the iv is larger than the previous message. Note that we divide this part by different sender. Since we not only need to decrypt messages from others, but we also need to decrypt messages sent by ourselves. We use a separate IV counter for decryption tracking.

Step4. Decrypt message.

Get the correct decryption key and use '**decrypt()**' to get plaintext.

Local storage:

The following chart shows the local storage after chatting. We can see that the corresponding keys are the same, and the iv tracker works well.

secretbits_USER2_COUNTER1	241,178,184,233,255,72,233,190,184,151,99,133,180,2...
keyset_counter_USER2_SECRET1	1
secret_counter_USER2	1
CHAT_KEY_USER1to2_KEYSET1_SECRET1	143,192,80,239,230,54,102,75,182,77,149,242,187,183,...
CHAT_KEY_USER2to1_KEYSET1_SECRET1	135,50,62,25,104,232,229,181,201,227,253,182,157,31,...
encryption_iv_USER2_KEYSET1_SECRET1	2
decryption_iv_USER2to1_KEYSET1_SECRET1	1
decryption_iv_USER1to2_KEYSET1_SECRET1	2
CHAT_MAC_USER1to2_KEYSET1_SECRET1	{"alg":"HS256","ext":true,"k":"j8BQ7-Y2Zku2TZXyu7dhi...
CHAT_MAC_USER2to1_KEYSET1_SECRET1	{"alg":"HS256","ext":true,"k":"hzI-GWjo5bXJ4_22nR-W...

secretbits_USER1_COUNTER1	241,178,184,233,255,72,233,190,184,151,99,133,180,2...
keyset_counter_USER1_SECRET1	1
secret_counter_USER1	1
CHAT_KEY_USER2to1_KEYSET1_SECRET1	135,50,62,25,104,232,229,181,201,227,253,182,157,31...
CHAT_KEY_USER1to2_KEYSET1_SECRET1	143,192,80,239,230,54,102,75,182,77,149,242,187,183...
encryption_iv_USER1_KEYSET1_SECRET1	1
decryption_iv_USER2to1_KEYSET1_SECRET1	1
decryption_iv_USER1to2_KEYSET1_SECRET1	2
CHAT_MAC_USER2to1_KEYSET1_SECRET1	{"alg":"HS256","ext":true,"k":"hzI-GWjo5bXJ4_22nR-...
CHAT_MAC_USER1to2_KEYSET1_SECRET1	{"alg":"HS256","ext":true,"k":"j8BQ7-Y2Zku2TZXyu7d...

(we start new conversations, this local storage is not related to the ones in the previous session)

3.2.4 Key refresh

All symmetric keys and IVs should be re-derived from the shared secret when user clicks on a "Refresh" button in the chat, using a new salt.

Workflow

Sender side

```
async function refreshKeys() {
    console.log('Refresh keys functionality will be implemented here.');
    await generate_edmac_key_user1();
    await sendRefreshKeyinit();
}
```

Receiver side

```
async function reply_to_key_refresh(message){
  console.log('Receive information to refresh keys');
  result = await verifyOldMac(message, "change");
  if (result==false ){
    window.alert("invalid old mac");
    return;
  }
  result2 = await verifyNewMac(message, "change");
  if (result2 == false){
    window.alert("invalid new mac");
    return;
  }
  await generate_edmac_key_user2();
}
```

Verify old MAC

Verify new MAC

Generate new keyset

For the Sender who sends the key refresh request, it derives a new key from the shared secret and sends the request to the other. The key derivation for encryption/decryption key pair and MAC keys are the same as stated in **3.2.2 key derivation**. The salt we used is the counter for the keyset, which will increase when generating a new keyset. To send the refresh key request, following the JSON format in the data structure before, we need authentication tags for the old MAC signature and the new MAC signature. We use similar functions stated in **3.2.3 Normal message sending and**

receiving Sender (b)Protect the IV with HMAC-SHA256 to sign the last iv altogether with string “change” as additional information. And put old mac in message_value and new mac in message_tag, which is consistent with the data structure.

First, the receiver needs to check the current time with the message created time (by default in the database) to see whether it is within 1 minute. Then, before generating new key pairs, the new MAC and old MAC must be verified first. The verification function is similar to **3.2.3 Step 2. Verify signature**. Since we store our keyset with both keyset counter and secret counter, getting the correct key pair for operation is straightforward.

After one key refresh, the screen will show “==Keys changed==.” The window will alert the error information if the signature is invalid. So, if there is a “keys changed” message with a window alert, someone sends an invalid key refresh request. Since the request is invalid, our key will not refresh in this situation.

Note that since the functions used in this key refresh are similar to those in the previous session, we don’t repeatedly show the code. You can either refer to the previous session or the source code.

The following image shows the local storage after the key exchange. You can compare this one with the figure in the previous session. Older keys are stored for older message encryption. We can see that the counter for the keyset increases by one while the secret counter is still the same (because we didn’t generate a new secret but derived new keys from the secret).

secretbits_USER1_COUNTER1	241,178,184,233,255,72,233,190,184,151,99,133,180,2...
keyset_counter_USER1_SECRET1	2
secret_counter_USER1	1
CHAT_KEY_USER2to1_KEYSET1_SECRET1	135,50,62,25,104,232,229,181,201,227,253,182,157,31...
CHAT_KEY_USER1to2_KEYSET1_SECRET1	143,192,80,239,230,54,102,75,182,77,149,242,187,183...
encryption_iv_USER1_KEYSET1_SECRET1	1
decryption_iv_USER2to1_KEYSET1_SECRET1	1
decryption_iv_USER1to2_KEYSET1_SECRET1	2
CHAT_MAC_USER2to1_KEYSET1_SECRET1	{"alg":"HS256","ext":true,"k":"hzI-GWjo5bXJ4_22nR-...
CHAT_MAC_USER1to2_KEYSET1_SECRET1	{"alg":"HS256","ext":true,"k":"j8BQ7-Y2Zku2TZXyu7d...
CHAT_KEY_USER2to1_KEYSET2_SECRET1	250,107,109,158,48,230,204,125,226,122,20,205,42,17...
CHAT_KEY_USER1to2_KEYSET2_SECRET1	75,86,125,195,50,122,88,98,161,248,199,72,34,27,74,2...
encryption_iv_USER1_KEYSET2_SECRET1	0
decryption_iv_USER2to1_KEYSET2_SECRET1	0
decryption_iv_USER1to2_KEYSET2_SECRET1	0
CHAT_MAC_USER2to1_KEYSET2_SECRET1	{"alg":"HS256","ext":true,"k":"-mttnjDmzH3iehTNKrF-...
CHAT_MAC_USER1to2_KEYSET2_SECRET1	{"alg":"HS256","ext":true,"k":"S1Z9wzJ6WGKh-MdIIht...

3.2.5 Local Storage

We store the encryption, MAC keys, and corresponding iv using local storage with HTML5 Local Storage API. This assures that the keys are within the range of convenience without losing the security aspect if the local environment is secure.

Besides the key materials and counters stated in the previous session, we also store the related information for old message decryption. The use of

associated variables will be explained in session **3.2.5 message history**.

3.2.6 Message History

Our chat interface shows the history of previous messages and the new messages. If Local Storage has been cleared, previous messages cannot be decrypted, and a warning will appear. The method we used to identify which set of keys to use for a given message is keeping track of the variables mentioned in the last session for old message decryption.

Last_response_USER(i): We store the created time of the last message we responded to and use it to check whether this message is old or new.
Valid_since_USER(i): This variable stores the earliest time a message can be decrypted. It is used to detect the undecryptable message caused by clearing local storage. It is reset during the ECDH initial request process because an ECDH request will only be sent when the local storage is cleared. Which means the previous message can not be decrypted. We record the time so that we can check whether related key materials exist for older messages. When receiving an ECDH request, we check whether local storage is cleared. If it is cleared, we also need to set the variable.
Secret_START_USER(i), keyset_START_USER(i), decryption_iv_START_USER(i)to(j): These counters record which secret, which keyset, and which iv to use for decrypting the first older messages. Once we know the information to decrypt the first message, we can get all the information for the following messages since we have the recording of the decryption vector for each key set and the keyset counter for each shared secret. The details of implementation will be introduced soon. Since these variables keep track of the info of the first old message, they are set in the process of ECDH request (ECDH response if local storage is null) and erase chat (the value of the variables are set to be the current values of the secret counter, keyset counter, and decryption ivs). The purpose is to know where to start when given a batch of old messages.

We added the erase chat message to our project to keep track of the above variables. When a user erases a chat, all messages related to him in the database will be deleted. So we need to keep track of the changes in the first

message of the old messages. It is necessary to notify the peer user that I erased the chat.

The core for decrypting older messages is determining which secret, keyset, iv to use. Since we only store the info to decrypt the first one in the old messages, we need variables in .js to keep track of the secret counter, keyset counter, and decryption ivs.

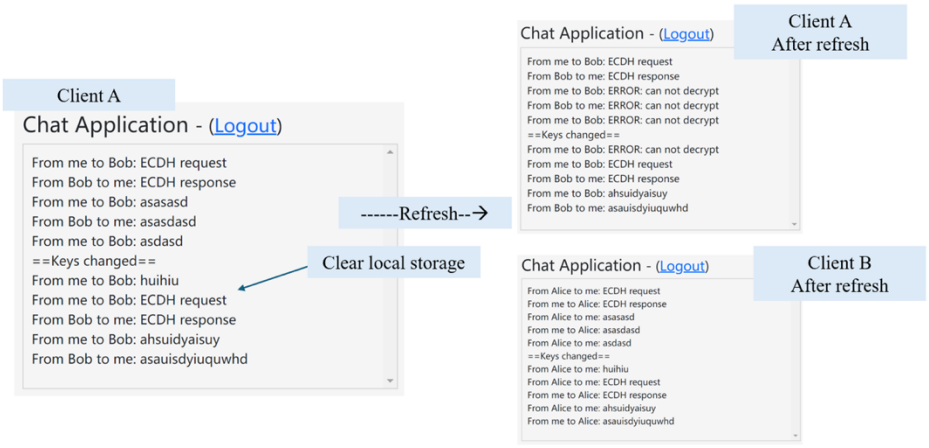
```
var decryption_iv_me_to_peer;
var decryption_iv_peer_to_me;
let s_secret;
let c_keyset;
```

The initial values of the above variables are set to Secret_START_USER(i), keyset_START_USER(i), and decryption_iv_START_USER(i)to(j). Once we get the info for the first message, we use logic like the addition with carry in math.

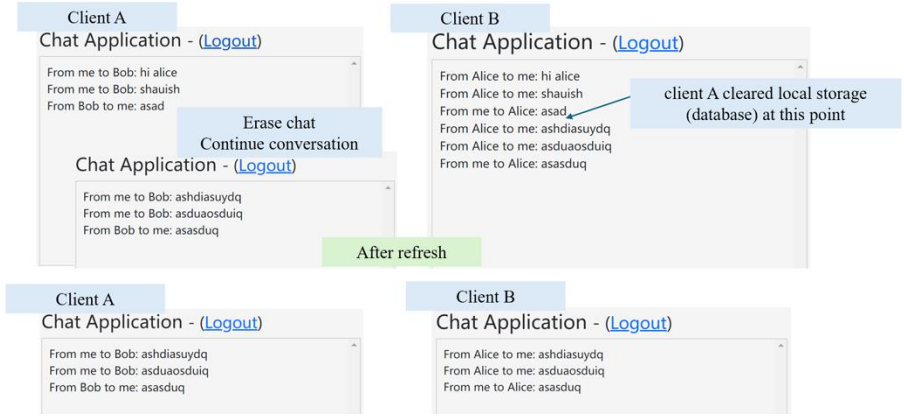
```
let secret_counter = JSON.parse(localStorage.getItem("secret_counter_USER"+peer_id.toString()));
let counter = JSON.parse(localStorage.getItem('keyset_counter_USER'+peer_id.toString()+"_SECRET"+secret_counter.toString()));
m2p = JSON.parse(localStorage.getItem("decryption_iv_USER"+myID.toString()+"to"+peer_id.toString()+"_KEYSET"+c_keyset+"_SECRET"+s_secret));
p2m = JSON.parse(localStorage.getItem("decryption_iv_USER"+peer_id.toString()+"to"+myID.toString()+"_KEYSET"+c_keyset+"_SECRET"+s_secret));
c = JSON.parse(localStorage.getItem('keyset_counter_USER'+peer_id.toString()+"_SECRET"+s_secret.toString()));
if (s_secret == secret_counter && c_keyset == counter && m2p == decryption_iv_me_to_peer && p2m == decryption_iv_peer_to_me){
    older=false;
    console.log("old messages finished.");
    return decryptMessage(message);
}
else if (c_keyset == c && m2p == decryption_iv_me_to_peer && decryption_iv_peer_to_me == p2m){
  console.log(decryption_iv_me_to_peer);
  console.log(decryption_iv_peer_to_me);
  s_secret = s_secret+1;
    c_keyset = 1;
    decryption_iv_me_to_peer = 0;
    decryption_iv_peer_to_me = 0;
}else if (m2p == decryption_iv_me_to_peer && decryption_iv_peer_to_me == p2m) {
  console.log("this enter");
    c_keyset = c_keyset+1;
    decryption_iv_me_to_peer = 0;
    decryption_iv_peer_to_me = 0;
}
```

If all the variables are the same as the current secret counter, keyset counter, and ivs, the decryption for old messages is finished. If the ivs and keyset counters are precisely the same as the value stored corresponding to that secret, all messages using that secret have been decrypted. We need to increase the secret number. If the IVs are the same, all messages using that keyset are decrypted. We need to increase the keyset counter and use the next keyset to decrypt all following old messages.

Experiment 1. The following experiment shows the performance of decrypting older messages. Client A will generate the ECDH request when the local storage is cleared. We send several messages to test it. Then, we reload the browser. We can see that since client A does not have the previous key materials, the older messages can not be decrypted. Client B still has the materials to decrypt the earlier messages. This shows the correctness of our implementation. The initial values of the above variables are set to Secret_START_USER(i), keyset_START_USER(i), and decryption_iv_START_USER(i)to(j). Once we get the info for the first message, we use logic like the addition with carry in math.



Experiment2. Check whether old messages can still be decrypted on both sides after erasing the chat (delete messages in the database). It works.



3.2.7 Key Exchange Initiation

The ECDH key exchange is triggered automatically. In our implementation, we check the local storage every minute to detect whether it is cleared.

The method we use is checking for a particular variable, **"USER(peer_id)."** This is used as a flag since it is stored with ECDH request and ECDH response (if local storage is cleared). We assume that all other key materials exist if it exists and vice versa.

We set the time interval to be one minute because this is a tradeoff between network traffic and efficient detection. The local storage will not often be cleared, so it is unnecessary to detect it so frequently, like every second. Imagine your local storage is cleared, and you want to chat with Bob, but Bob is not online. Then, if you set the time interval to be short, it will continuously send ECDH requests to Bob; this is annoying, messy, and useless.

The case for 1-minute intervals is that if you clear the local storage while chatting, you need to wait till the detection. Since the detection interval is 1

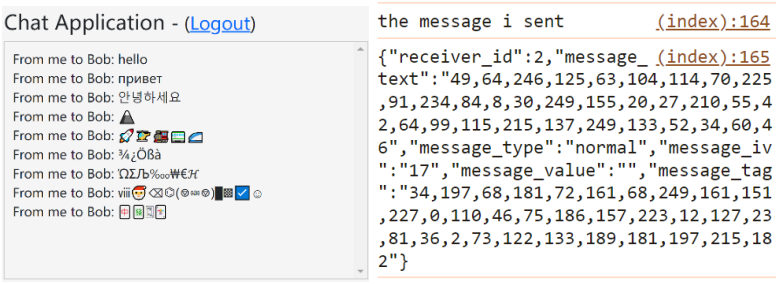
minute, on average, you need to wait for 30 seconds, and 60 seconds is the worst case. But usually, people will not clear the local storage while chatting with others. They typically clear it after a conversation or when not using the application. This may not be an issue due to the average human behavior.

Race case problem solving: considering both clients have empty local storage, we need to solve the problem of them sending the ECDH request simultaneously. We set the rules that the user with a smaller ID will always send the ECDH request first. Since we can not modify the protocol, we implement this by letting the user with a larger ID wait for 3 seconds; if there are no ECDH requests, it will send the request. This may work in most cases, but theoretically, there is a very low probability that there could still be a conflict. But during our test, this situation never occurred.

3.2.8 Encode chat messages and Format network message

The encoding in UTF-8 is applied to each chat message to work with different kinds of characters and languages. JSON format network messages respect our custom schema following the goal of efficient parsing and handling of chatting structures. You could refer to the data structure at the beginning of the E2EE part for details.

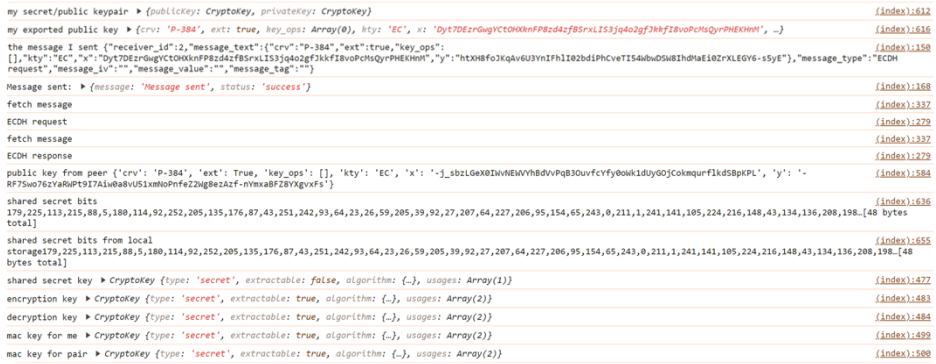
(The message on the right corresponds to the last message sent on the left.)



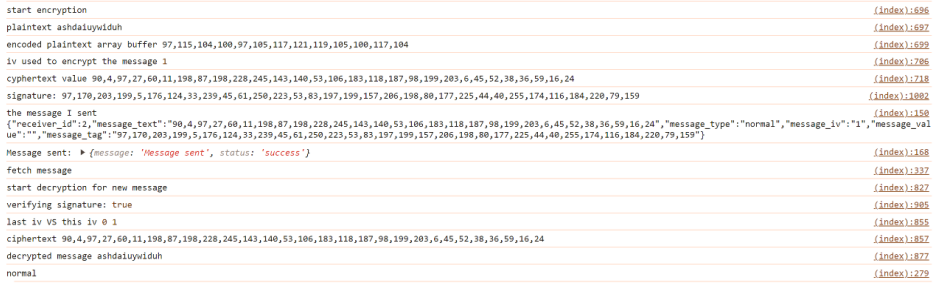
3.2.9 Logging

A console.log() is used to log cryptographic operations for debugging and development. This enables audit and fault tracing for security operations, simplifying auditing and problem-solving. You can check the logging by building our project. Some of the logs were already used in the previous session. We show some examples below.

Example. 1 ECDH key exchange log



Example 2. message encryption and decryption log



3.2.10 Security Protections -> 3.1.8 Prevention of XSS, CSRF and SQL injection

3.3 TLS

3.3.1 Generate a TLS key and CSR

Considering the task requirements, we launched the job which gave us an access to OpenSSL- a reliable free software for managing TLS and SSL protocols. Using SHA-384 we have generated private key with ECC crypto-system over P-384 elliptic curve for CSR submission. In this regard privacy protection is a key issue to be taken care of. This way, the implementation of the latest cryptographic standards will be ensured thus providing a high level of security and data secrecy.

3.3.2 Customize the OpenSSL configuration file

The openssl.cnf configuration file was much altered to include many of the domain names for the certificates. To align with the project's requirements, we incorporated several critical settings: check-box is set to off and Key Usage specified as Digital Signature. Moreover, we assign Extended Key Usage for Server Authentication and finally, we alter the validity period to 3 months. Modifications were key to ensure the customization of the certificate to really suit sectors of security and their operational needs.

```
[ req ]
distinguished_name  = req_distinguished_name
policy              = policy_match
x509_extensions     = user_crt
req_extensions      = v3_req

[ policy_match ]
countryName             = match  ## Must match RootCA CSR and any certificate this RootCA signs
commonName              = supplied

[ req_distinguished_name ]
countryName                     = Country Name (2 letter code)
countryName_default             = HK
countryName_min                 = 2
countryName_max                 = 2
commonName                      = Common Name (eg, your name or your server hostname) ## Print this message
commonName_default              = group-12.comp3334.xavier2dc.fr
commonName_max                  = 64

[ user_crt ]
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid,issuer
basicConstraints        = critical, CA:FALSE
keyUsage                = critical, digitalSignature
extendedKeyUsage        = serverAuth
subjectAltName         = DNS:group-12.comp3334.xavier2dc.fr

[ v3_req ]
basicConstraints        = critical, CA:FALSE
extendedKeyUsage        = serverAuth, clientAuth, codeSigning, emailProtection
keyUsage                = nonRepudiation, digitalSignature, keyEncipherment
```

3.3.3 Generate the certificate

By using a pre-existing root certificate and key, we completed the CSR that we had before. It was the logical conclusion to the undertaking, and this certificate is specifically for the web application.

The command used for this process was:

openssl x509 -req -in group12.csr -CA cacert.crt -CAkey cakey.pem -CAcreateserial -out group12.crt -days 90 -SHA384 -extensions user_crt -extfile openssl.cnf

The certificate is shown as follows:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            12:d6:a7:9a:5a:26:fb:7c:ef:2d:5f:6f:93:b4:db:4f:88:c5:36:c9
        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: Mar 28 04:52:35 2024 GMT
            Not After : Jun 26 04:52:35 2024 GMT
        Subject: C = HK, CN = group-12.comp3334.xavier2dc.fr
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:b3:ae:89:4b:a4:75:30:06:b1:85:40:68:b3:b9:
                    12:0e:a1:aa:55:d6:f4:fa:cf:b5:95:0f:c9:f3:3d:
                    55:cf:a1:35:92:c7:52:0a:e8:55:ab:1c:ac:9e:e3:
                    8a:ce:cc:b7:64:30:34:d3:06:f4:2b:68:e8:31:d3:
                    14:c2:b9:51:2c:57:57:19:8f:ea:4e:b1:97:56:69:
                    b5:ac:dc:54:f5:2a:b8:ac:40:a5:45:fa:54:57:81:
                    8c:61:2b:c3:e3:6c:5a
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                3C:43:02:EF:4E:C4:60:A7:A0:E7:F3:43:4F:B4:9B:10:4A:DF:2E:F4
            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
            X509v3 Subject Alternative Name:
                DNS:group-12.comp3334.xavier2dc.fr
    Signature Algorithm: ecdsa-with-SHA384
         30:81:88:02:42:01:fd:83:94:78:b7:e3:3d:ee:7e:4d:1c:22:
         a3:ea:ec:f3:d8:14:ee:f9:8b:11:a5:55:b0:24:c1:15:7b:e3:
         40:63:1d:f1:67:0f:f9:54:3f:55:d1:44:78:29:7a:5e:70:31:
         89:2c:90:a3:15:3a:5a:10:ea:ed:62:be:8a:62:38:72:8e:02:
         42:01:1a:af:cd:e5:79:21:e7:1e:de:e5:43:87:d5:7b:6f:4f:
         f1:62:41:c5:b7:fb:54:0d:b1:cb:36:88:f5:ed:ec:8a:35:dd:
         7c:19:ee:94:b8:ad:29:2e:0d:5c:ce:4f:c3:ad:97:dd:33:f5:
         b2:3b:6a:1b:41:22:d9:f4:c4:93:5c:9b:dd
```

This command checks signing and gives the certificate a temporal period with required cryptographic hash function, as well the particular extensions.

3.3.4 Configure nginx and docker file of the web application

For this part, we change the original setting of the Nginx file and docker compose file: firstly, we add a ssl config to the nginx file and then we put the certificate and the private key we generated previously into the file and add two paths to the nginx file to make sure they can be used. After that, we use Mozilla's "modern" configuration for nginx that is given in the project and add the rest of requirements of the project for nginx settings, including change the port to 8443 as the requirements needs and that help us to secure the website to HTTPS with the certificate and key that we generated. And finally add a path for certificate and keys in the docker.yaml file.

```
server {
    listen 8443 ssl;
    server_name group-12.comp3334.xavier2dc.fr;

    # SSL configuration
    #
    ssl_certificate group12.crt;
    ssl_certificate_key ec384-key-pair-group12.pem;
    ssl_session_timeout 1d;
    ssl_session_cache shared:MozSSL:10m;  # about 40000 sessions
    ssl_session_tickets off;

    # modern configuration
    # requirement: 3.1
    ssl_protocols TLSv1.3;
    ssl_prefer_server_ciphers off;

    # HSTS (ngx_http_headers_module is required) (63072000 seconds)
    # requirement 3.5
    add_header Strict-Transport-Security "max-age=604800" always;

    # requirement 3.3
    # ssl_ciphers TLS_CHACHA20_POLY1305_SHA256;
    ssl_conf_command Ciphersuites TLS_CHACHA20_POLY1305_SHA256;
    # requirement 3.2
    ssl_ecdh_curve X25519;
```

```
nginx:
  image: nginx:1.25.4
  ports:
    - 8080:8080
    - 8443:8443
  volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf
    - ./ec384-key-pair-group12.pem:/etc/nginx/ec384-key-pair-group12.pem
    - ./group12.crt:/etc/nginx/group12.crt
  depends_on:
    - webapp
```

3.3.5 Test the application

After the previous steps, we tested the application and found that it is secured with HTTPS.

4. Autonomy and Creativity

4.1 Image-based CAPTCHA & Text-based CAPTCHA

4.1.1 Autonomy

Text-based CAPTCHA:

High Autonomy in Implementation: The creation of the text-based CAPTCHA involved an all-around rather learning the JavaScript language to integrating the capture into the logic page in HTML. On this way, developer should be given a greater freedom, as we were deeply involved in each CAPTCHA stage - starting from generating the CAPTCHA to validating it at the server side.

Custom Solution: It has been shown by CAPTCHA flexibility to construct and implement it as needed in a specific situation. We used the script to build up

CAPTCHA, with particular attributes, including symbol set, length, and random selection, which contain a customized solution targeted to the app's needs.

Image-based CAPTCHA:
Limited Autonomy: Deployment of hCaptcha for pictures with CAPTCHA entails a dependence on an outsource tool now. The CAPTCHA code generation, validation, and updates are all depended upon the external mechanism; therefore, there is no authenticity here.

Ease of Integration: However, it diminishes autonomy, when the hCaptcha is set up. Then, it eliminated the need for high integration procedures. Developers need to demonstrate only small effort in order to interface with the service by embedding it into their webpage and API calls become their only job to check the verification. It is a much effortless and comfortable way than to develop a system from scratch

4.1.2 Creativity

Text-based CAPTCHA:
Creative Control: We may try to replicate the most advanced and still free CAPTCHA variations, since there are no limits to how one can design a CAPTCHA. These encompass aspects such as the visual display aesthetic of artwork on a canvas through to CAPTCHA complication ability which can be moderated to meet the end user demands and security.

Innovative Implementation: Both the creation and the rendering rely on cutting-edge coding technics in JavaScript which allow for a CAPTCHA service that is unique and suitable both for a website appearance and functional needs.

Image-based CAPTCHA:
Standardized Creativity: Image-based CAPTCHA with hCaptcha is a creative tool if we regard it as a case when creative scope provided by the service is approximately limited to the customization options. These can range from desktop versions to those optimized for mobile experiences. We can also include some levels of visual customizations for which to match the website while maintaining a distinct brand identity.

Focus on Functionality Over Design: The function and security are given priority considering that we work more now for industrial purposes and maintain a low budget. However, the questions generated by the hCaptcha system are static in nature and tailoring the design results in minor customization.

4.1.3 Conclusion

Text-based CAPTCHA plays a major role in the process of autonomy and creativity. The text-based system offers more freedom in appearance and security issues, the developers being able to customize both the overall look and the particular security features for a desired output. Compared to the textual CAPTCHA, image-based CAPTCHA, even though it is easier to implement, and the development effort is relatively low, it comes with a very limited creative options and auxiliary resources as it heavily depends on the third parties' predefined capabilities and overall design goals.

4.2 Fetch message interval

We found that the setInterval() function is not synchronous during our implementation. Eg. For setInterval(function(), 1000), no matter whether function() will be finished in 1 second, the following parallel function() will be executed. If the variable used in the later execution depends on the previous result after the function, this may cause an error. By default, we use setInterval() to fetch messages every second. Considering that there are computers that are pretty slow so that the asynchronous situation may occur, we set the interval to be 2 seconds to prevent most of the cases from leading to a conflict.

4.3 Utils function

In our implementation, several utils functions are used to convert variables' format/data type to make them valid parameters while using the API. For example, the function to convert a number into 96 bits valid iv. These helper functions can be found in the last part of the chat.html.

5. Conclusion and reflection:

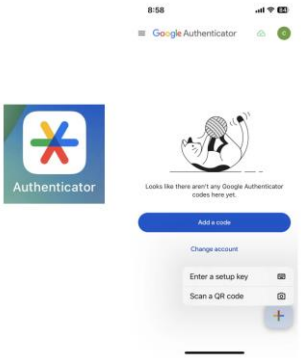
In this COMP3334 group project, we adapted a basic chat web application to become a secure E2EE chat web app, Complying with some of the requirements in NIST Special Publication 800-63B “Digital Identity Guidelines – Authentication and Lifecycle Management" for US federal agencies. A secure MFA mechanism based on passwords and OTP was used, and encrypted communications between two users were implemented. The communication is protected in transit by configuring a modern TLS deployment.

Through the group project, we learned a lot by implementing the project requirements. This is the first time we have learned JavaScript and implemented a full-stack project with Flask Python. By reading the NIST guideline, we realize that to implement a “secure” application, the things we need to consider are more than what we have learned in class before. Now, we better understand what aspects should be taken seriously. After implementing the E2EE, we genuinely use what we learned in class and apply the knowledge to practice. With the theory we learned from the textbook and the code implementation, we learned a lot from this subject, which could be significant for our future career and life-long study.

APPENDIX:
Instructions to use the application.
Registration:

1. Click the register link at the bottom of the login page to create an account.
2. Choose username and password. The follows provide **suggested** username and password, if you have trouble thinking of a charming name and valid password.
   - Username: Alice password: alicecomp3334
   - Username: Bob password: bobcomp3334
3. Click register button. You will see the following page. You are suggested to **take a photo** of this page, because the strings below the QR code could be further used for recovery secrets.

4. Bind authentication:
   - We make use of Google Authenticator APP. You first need to download the google authenticator from app store on your mobile device. Then click **add a code** and choose **scan a QR code**. Or you click the + button at the right bottom of the screen and chose **scan a QR code**. You can now scan the QR code to bind with your mobile phone.



   - After scanning, you will see an OTP corresponding to your name and COMP3334 Group12 on google authenticator.

   - After binding, click the **back** button. You will see the following notification.

Login:

Enter your username and password. You need to check the google authenticator for OTP. Since our OTP is time based. It will change for a given time interval. Make sure that your OTP is valid when you click the login button, otherwise there will be a notification for invalid OTP.