Project Report: SecureMessenger - A Secure Messaging Application

Author: Tejaswi Velaga **UIN:** 663404952

1. Introduction

SecureMessenger is a secure, web-based messaging platform built to simulate Telegram's dual-chat architecture. It supports two communication modes: Secret Chats and Cloud Chats. Each mode is built to reflect different security philosophies — one prioritizing end-to-end privacy, and the other offering server-side synchronization for usability. This report explains the design, implementation, and cryptographic principles used in each mode in a clear and beginner-friendly manner.

Key Features

SecureMessenger provides a flexible messaging experience through two primary communication modes: Secret Chats and Cloud Chats. Secret Chats offer true end-to-end encryption where only the sender and receiver can access message content. These chats rely on client-side encryption using Diffie-Hellman key exchange and AES-256-IGE, ensuring that the server has no visibility into the communication.
In contrast, Cloud Chats are designed for convenience and device synchronization. They use transport-level AES encryption, and messages are decrypted and temporarily stored on the server before being delivered. This model mimics Telegram's cloud-based messaging behavior.

The application also incorporates **Forward Secrecy**, a feature that periodically regenerates encryption keys in Secret Chats. This ensures that even if a key is later compromised, past messages remain protected.
Additional functionalities include real-time messaging over WebSockets and an in-memory message queue for offline delivery, all implemented with strict client-side cryptographic processing to maximize security and privacy.

2. Tech Stack and Cryptographic Assurance

Frontend:

- **HTML + Tailwind CSS**: Used to build a clean, responsive UI with minimal dependencies.
- **Vanilla JavaScript**: All client-side logic including WebSocket communication, key generation, and encryption/decryption is implemented using JavaScript.

Backend:

- **Node.js (TypeScript)**: The server is built using Node.js for handling WebSocket connections, message routing, session management, and cryptographic functions.

- **WebSocket (ws module)**: Real-time communication between clients and the server is handled using WebSockets for both Secret and Cloud Chat modes.

Cryptography:

- **Node Crypto Module**: Core cryptographic operations are powered by the built-in `crypto` module in Node.js, ensuring performance and security.
  - `crypto.createDiffieHellman()` – secure Diffie-Hellman key exchange
  - `crypto.createHash('sha256')` – for hashing shared secrets and generating msg_keys
  - `crypto.createCipheriv()` / `crypto.createDecipheriv()` – AES-256 encryption/decryption
  - `crypto.randomBytes()` – generates salts, IVs, and keys
- **AES-256-IGE Mode**: Simulated using AES-256-CBC logic with additional transformations to replicate IGE chaining, as used in Telegram.
- **KDF (Key Derivation Function)**: SHA-256-based derivation of AES keys and IVs from the combination of msg_key and the shared key.
- **Client Folder**: Contains HTML/CSS/JS assets and mode-specific UI rendering.
- **Server Folder**: Divided into multiple modules (`storage.ts`, `cloud_handler.ts`, etc.) to manage different chat modes cleanly.

3. Security Concepts

Data Integrity

- **Message Key Verification**: After decryption, the message hash (`msg_key`) is recomputed and matched to detect tampering.
- **Sequence Numbers & Timestamps**: Built into the payload to preserve message order and prevent replay attacks.

Authorization

- **Auth Key (Cloud Chat)**: A session-specific 256-bit key ensures message access is restricted to authorized sessions.
- **Auth Key ID**: A 64-bit derived ID enables fast lookup of the auth key.

Confidentiality

- **AES-256 Encryption**: Used in both chat modes.
- **Key Derivation**: SHA-256-based derivation ensures unique AES keys per message.

End-to-End Encryption (Secret Chats)

- Messages are decrypted only on the client.
- Server never has access to keys or plaintext.
- Uses `key_fingerprint` to validate correct key usage.

Forward Secrecy

- To enhance long-term security and prevent retroactive message compromise, SecureMessenger implements a key rotation strategy in its Secret Chat mode. Each key is monitored for usage, and once it has been used to encrypt and decrypt more than 100 messages, or has been active for over one week, a re-keying process is

initiated. This ensures Perfect Forward Secrecy (PFS), meaning that even if a future key is compromised, previous communications remain secure. Old keys are securely discarded and cannot be reconstructed from the newly established keys.

4. Message Payload Construction

Secret Chat Payload Structure:

- Fields:
  - **Length (32-bit)**
  - **Payload Type (32-bit)**
  - **Random Bytes (min 128-bit)**
  - **Layer (32-bit)**
  - **IN_seq_no / OUT_seq_no (32-bit)**
  - **Message Type and Object** (variable length)
  - **Padding (12–1024 bytes)**
- msg_key = SHA-256(payload + auth fragment)
- AES-IGE encryption applied using derived keys

Cloud Chat Payload Structure:

- Fields:
  - **Salt (64-bit)**
  - **Session_id (64-bit)**
  - **Payload (variable)**
  - **Padding (12–1024 bytes)**
- Encrypted with AES-IGE using session-based keys
- Sent as: auth_key_id, msg_key, and encrypted data

5. Secret Chats

5.1 What Are Secret Chats?

Secret Chats use **end-to-end encryption (E2EE)** to ensure that only the sender and receiver can read the message. The server cannot decrypt or see the message content.

5.2 Architecture

Secret Chat is architected with a client-centric security model, where encryption and decryption occur only on user devices. The server has no visibility into the content or keys.

Client Side:

- Each client generates a Diffie-Hellman key pair (public and private).
- Public keys are exchanged via the server.
- A shared secret key is derived from both public-private key pairs.
- Messages are serialized into a structured payload with metadata (layer version, sequence numbers), content, and padding.
- A SHA-256 hash of the payload is used to compute a msg_key, from which AES-256 keys and IVs are derived.

- The payload is encrypted using AES-IGE.
- Final message structure includes: key_fingerprint, msg_key, and encrypted payload.

Server Side:
- Relays public keys and encrypted messages between users.
- Does not decrypt or analyze the content.
- Temporarily queues messages in memory if the recipient is offline.

This design ensures complete end-to-end encryption, making the server a stateless router without any decryption capability.

5.3 Message Flow
1. DH key exchange between clients
2. Payload created and encrypted on sender's side
3. Message sent to and forwarded by server
4. Recipient decrypts using the same shared key

5.4 Security Summary
- Full end-to-end encryption
- Zero server knowledge

6. Cloud Chats

6.1 What Are Cloud Chats?
Cloud Chats prioritize usability and multi-device sync. Messages are encrypted in transit but decrypted and stored on the server.

6.2 Architecture
Cloud Chat architecture reflects a server-centric design where the server temporarily handles decrypted messages to support features like message history, sync across devices, and delivery to offline users.

Client Side:
- After user login, a persistent session key (auth_key) is generated through a Diffie-Hellman exchange.
- Each outgoing message is wrapped with metadata such as salt, session ID, and message body.
- A msg_key is calculated using SHA-256 over the payload.
- Using the msg_key and auth_key, the client derives AES-256 encryption keys and IVs via a KDF.
- The message is encrypted using AES-IGE and sent to the server with its auth_key_id and msg_key.

Server Side:
- The server uses auth_key_id to retrieve the correct session key.
- Decrypts the message and verifies its msg_key for integrity.
- Stores or relays the message as necessary.

- When delivering to the recipient, the message is re-encrypted using the recipient's session key and forwarded.

This model emphasizes usability by enabling features like chat history and offline message retrieval, though it comes with trade-offs in confidentiality compared to Secret Chats.

6.3 Message Flow

1. Client constructs payload and encrypts
2. Message sent to server
3. Server decrypts and stores
4. Server forwards to recipient
5. Recipient decrypts using own auth key

6.4 Security Summary

- Encrypted transport but server access to plaintext
- Easier multi-device support

7. Secret Chat vs. Cloud Chat – A Comparison

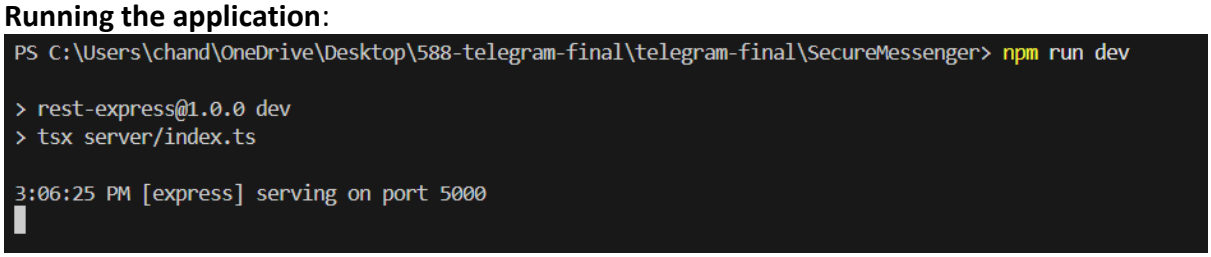
Feature	Secret Chat	Cloud Chat
Encryption Type	End-to-End (AES-IGE)	Transport (AES-256)
Server Decryption	No	Yes
Message Storage	Encrypted blob (in memory)	Plaintext (in memory)
Key Storage	Client only	Server + Client
Message Decryption Location	Client	Server, then Client

8. Design Philosophy and Justification

- **Modular Design**: Separate logic for chat modes avoids conflict.
- **No Persistent Storage**: In-memory message handling for simplicity and privacy.
- **Aligned With MTProto**: Payloads, hashing, and encryption follow Telegram's principles.

9. Output Screenshots

1. **Running the application**:



2. **User login- 2 users used(Alice and Bob)**

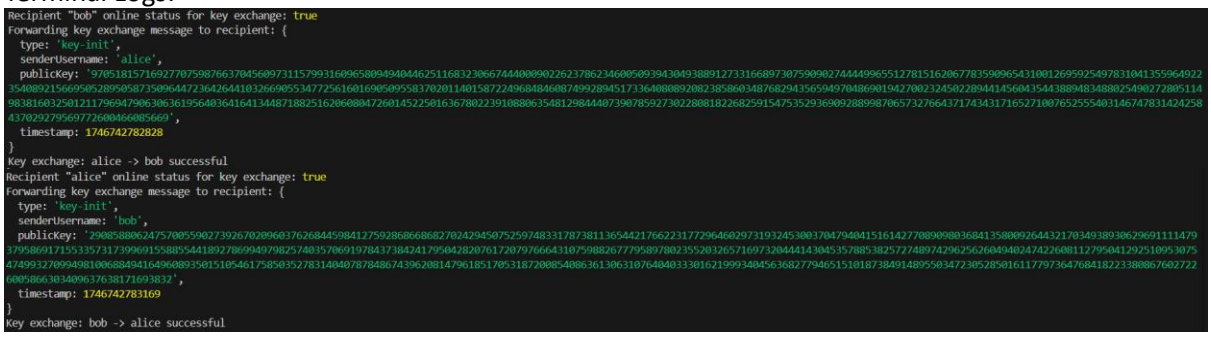
```
New WebSocket connection
Received WebSocket message: {"type":"login","username":"alice"}
Parsed message type: login
User alice added to storage. Total users: 1
User logged in: alice with cloud session e3ec3b99
New WebSocket connection
Received WebSocket message: {"type":"login","username":"bob"}
Parsed message type: login
User bob added to storage. Total users: 2
User logged in: bob with cloud session 28f3f821
```

3. **Secure connection Establishment between Alice and Bob for Message exchange**

Logs on UI:

```
> [5:17:22 PM] Establishing secure WebSocket
connection to ws://localhost:5000/ws
> [5:17:23 PM] WebSocket connection established
> [5:17:23 PM] Initial key pair generated: g^a mod
p = 4ce13da736...
> [5:17:23 PM] Received unknown message type:
session-created
> [5:19:42 PM] This is a secure private connect
with bob
> [5:19:42 PM] Sent key exchange to bob
> [5:19:43 PM] Received key exchange from bob
> [5:19:43 PM] Established shared key with bob
> [5:19:48 PM] This is a secure private connect
with bob
```

Terminal Logs:

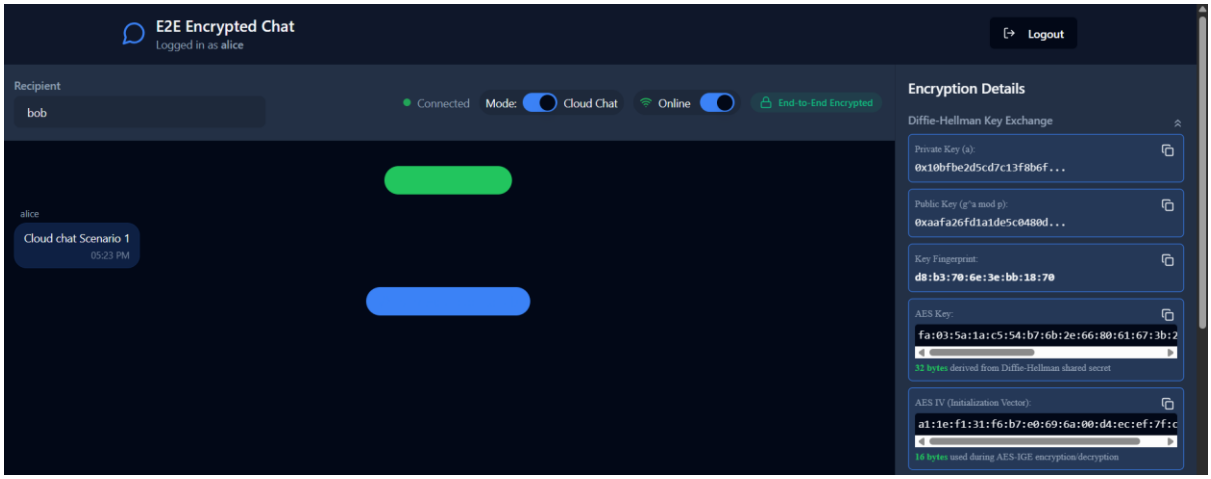


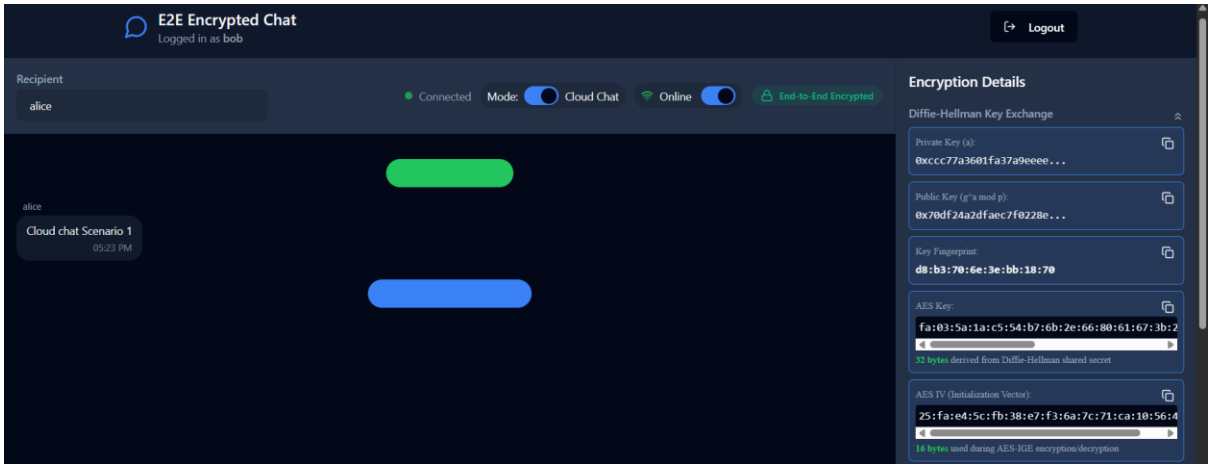
4. **When the sender is offline, message is queued until sender comes online:**

Sender Logs:

```
> [5:57:37 PM] Attempting to send message to bob
> [5:57:37 PM] User offline: Message queued for later delivery (hii)
> [5:57:38 PM] This is a secure private connect with bob
> [5:57:41 PM] Setting status to online
> [5:57:41 PM] Attempting to deliver 1 pending messages
> [5:57:41 PM] Sent queued message: "hii"
> [5:57:41 PM] Status update confirmed: online
> [5:57:41 PM] Received message from alice
> [5:57:41 PM] Received message: "hii"
```

5. **Cloud chat- Scenario 1: Both Alice and bob are online**





Encryption Logs:

```
[5:23:30 PM] [CloudChat] Mode: Cloud Chat
[5:23:30 PM] [CloudChat] Sending plaintext message: "Cloud chat Scenario 1"
[5:23:30 PM] [CloudChat] Encrypting using auth_key_id
[5:23:30 PM] [CloudChat] Derived msg_key: 861e80180acb87f4b883aa14c2a01c79
[5:23:30 PM] [CloudChat] AES-256 encryption complete.
[5:23:30 PM] [CloudChat] Encrypted payload sent to server.
> [5:23:32 PM] This is a secure private connect with bob
```

Decryption Logs:

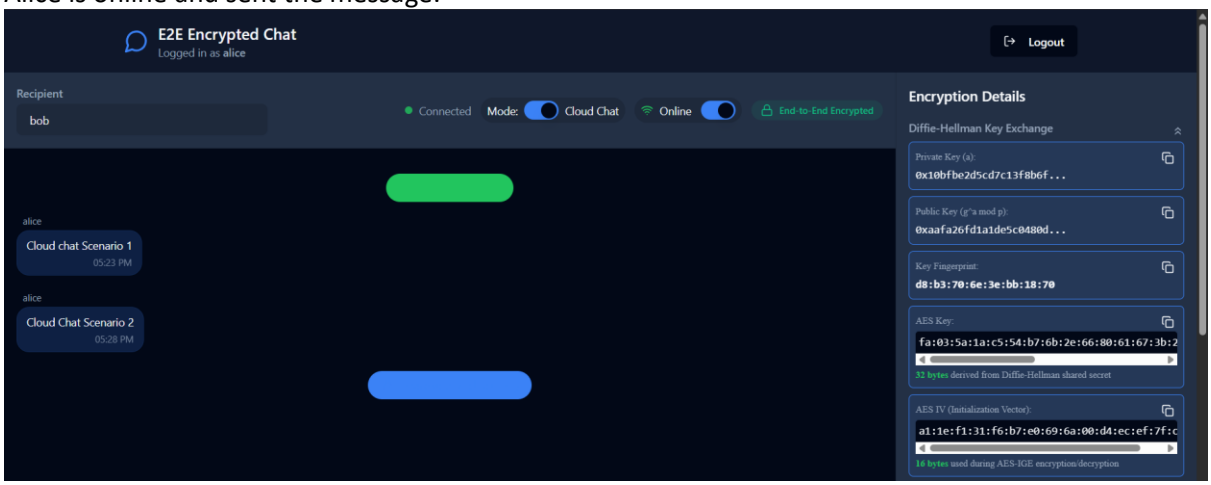
```
with alice
[5:23:30 PM] [CloudChat] Received encrypted message from user: alice
[5:23:30 PM] [CloudChat] Derived AES key/IV using msg_key: 861e80180acb87f4b883aa14c2a01c79
[5:23:30 PM] [CloudChat] Decryption successful. Plaintext: "Cloud chat Scenario 1"
[5:23:30 PM] [CloudChat] Storing message in cloud memory (to: bob)
```

Terminal Logs:

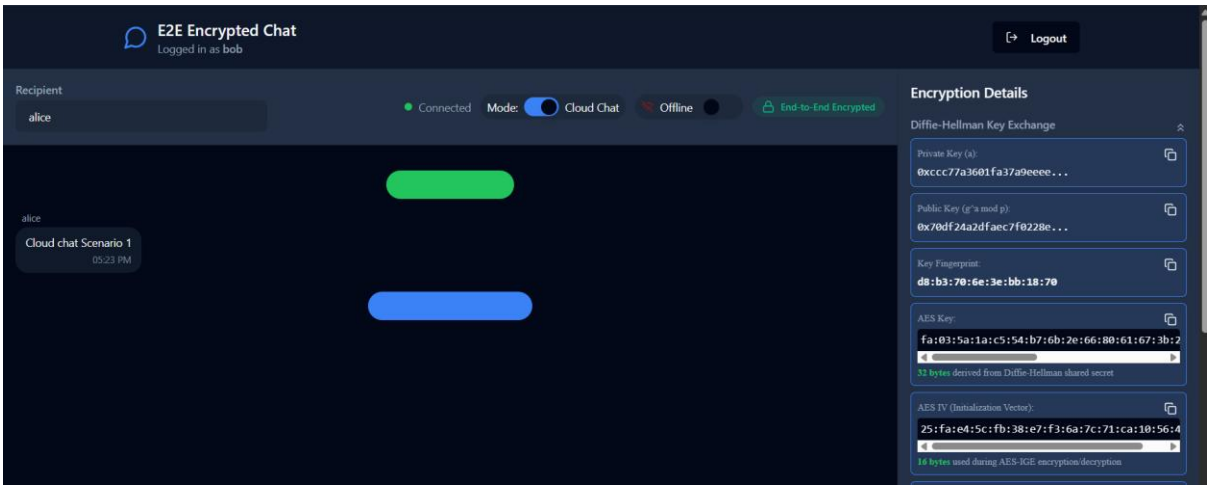

```
Received WebSocket message: {"type":"cloud-message","cloudMessage":{"msg_id":"cloud-1746743010030","seq_no":103920,"auth_key_id":"","msg_key":"861e80180acb87f4b883aa14c2a01c79","from_id":"alice","to_id":"bob","timestamp":1746743010030,"content":"Cloud chat Scenario 1","isCloudMessage":true}}
Parsed message type: cloud-message
[CLOUD CHAT] Processing message on server
Invalid session
[CLOUD CHAT] Message processed and stored on server
[CLOUD CHAT] Delivering message to online recipient bob
Received WebSocket message: {"type":"check-user","username":"bob"}
Parsed message type: check-user
User status check: bob is online
```


6. **Cloud chat- Scenario 2: Alice is online and bob is offline**

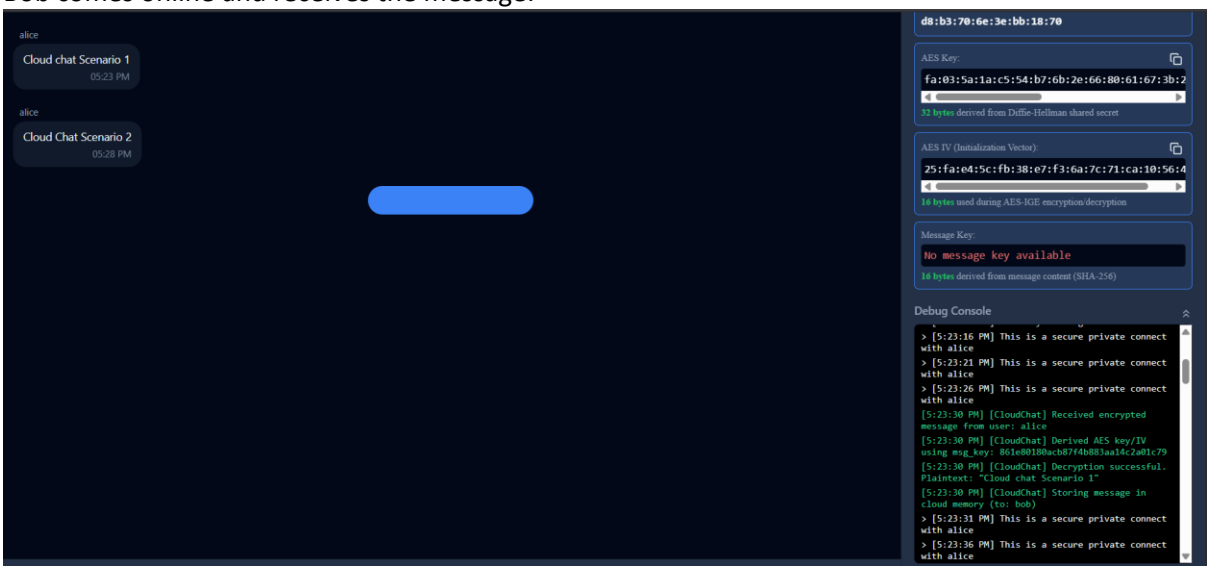
Alice is online and sent the message:



Bob is offline”



Bob comes online and receives the message:



Terminal logs:

Encryption logs on UI:

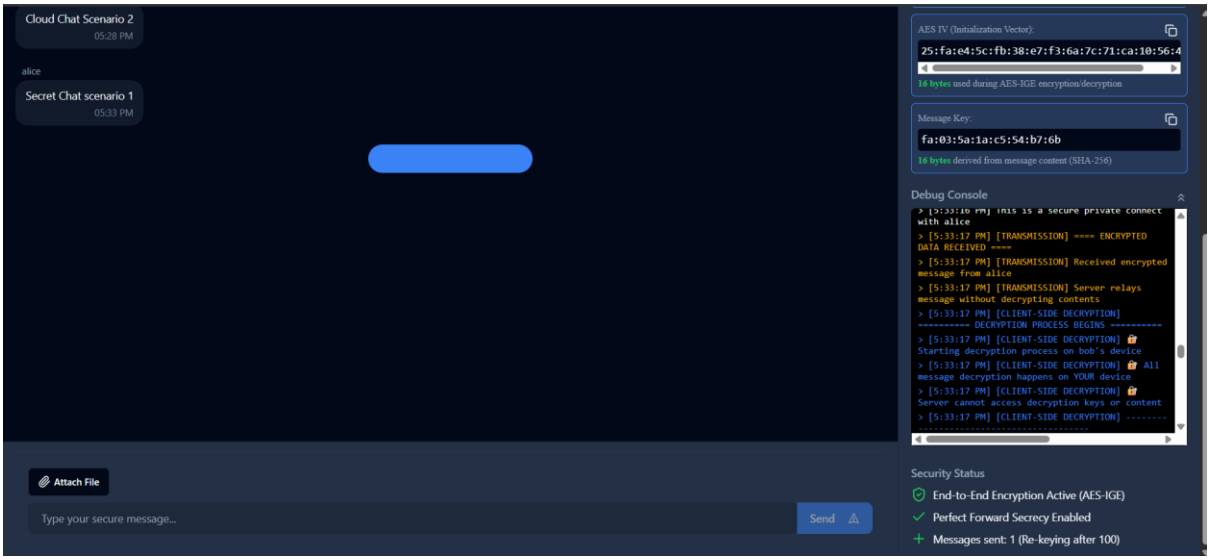
```
> [5:28:00 PM] Attempting to send message to bob
[5:28:00 PM] [CloudChat] Mode: Cloud Chat
[5:28:00 PM] [CloudChat] Sending plaintext
message: "Cloud Chat Scenario 2"
[5:28:00 PM] [CloudChat] Encrypting using
auth_key_id
[5:28:00 PM] [CloudChat] Derived msg_key:
c6009f8a3b722e61e111ffdeac7a6c2c
[5:28:00 PM] [CloudChat] AES-256 encryption
complete.
[5:28:00 PM] [CloudChat] Encrypted payload sent
to server.
> [5:28:02 PM] This is a secure private connect
with bob
```

Decryption logs on UI(bob's side):

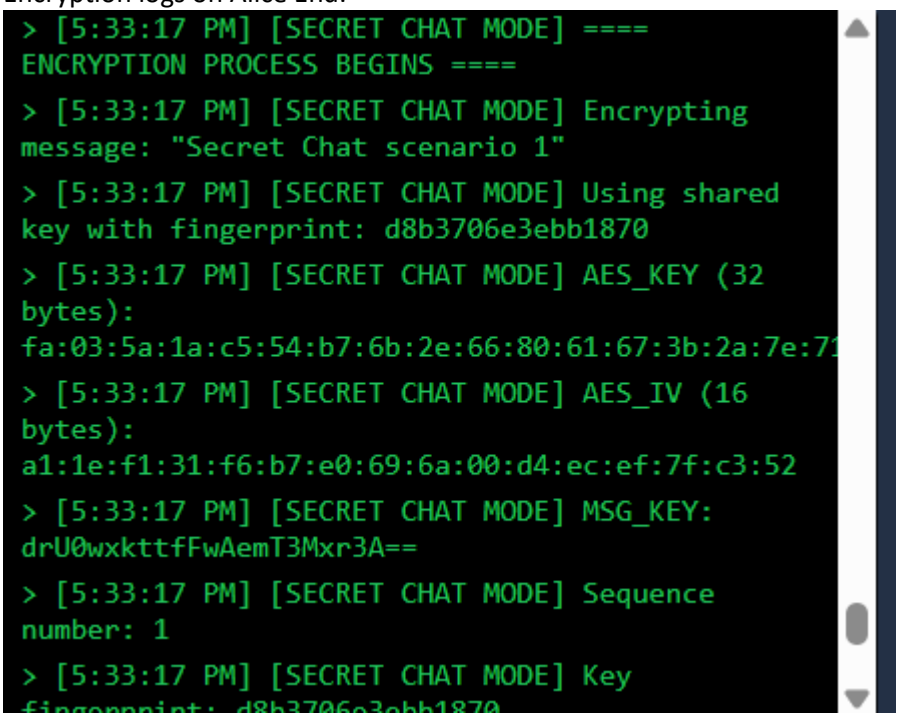
```
[5:28:31 PM] [CloudChat] Storing message in
cloud memory (to: bob)
[5:28:31 PM] [CloudChat] Received encrypted
message from user: alice
[5:28:31 PM] [CloudChat] Derived AES key/IV
using msg_key: c6009f8a3b722e61e111ffdeac7a6c2c
[5:28:31 PM] [CloudChat] Decryption successful.
Plaintext: "Cloud Chat Scenario 2"
[5:28:31 PM] [CloudChat] Storing message in
cloud memory (to: bob)
> [5:28:31 PM] Status update confirmed: online
```

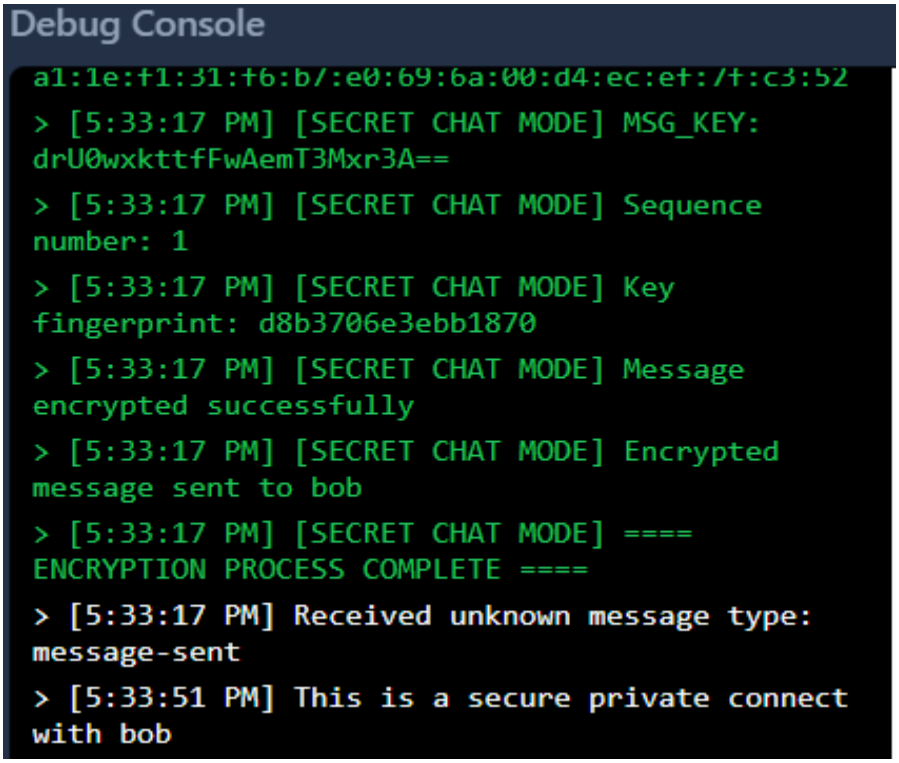
Secret Chat

Scenario 1: Alice and Bob are online

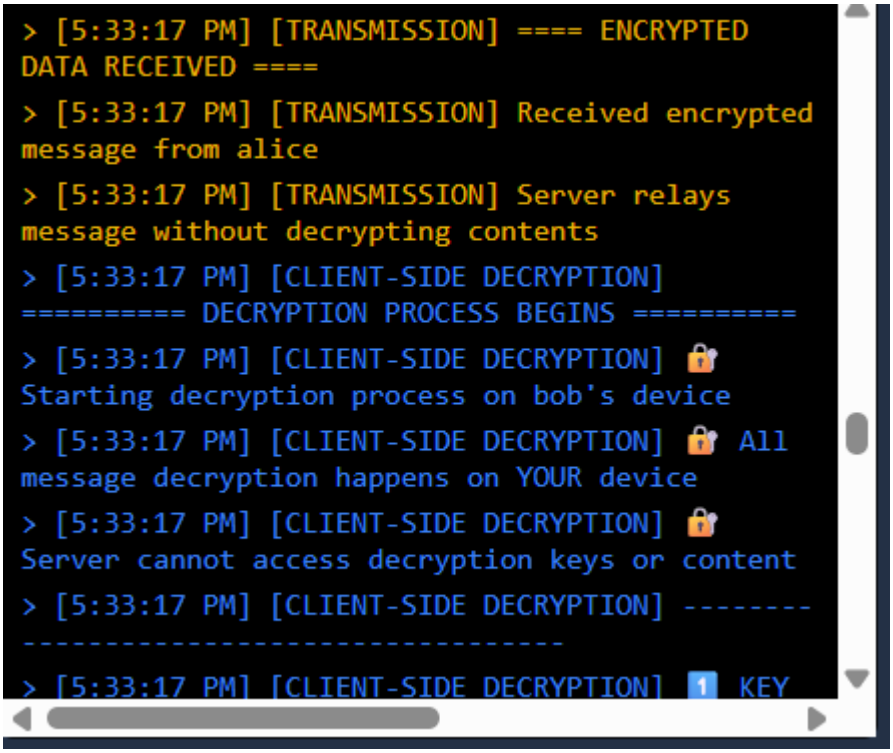


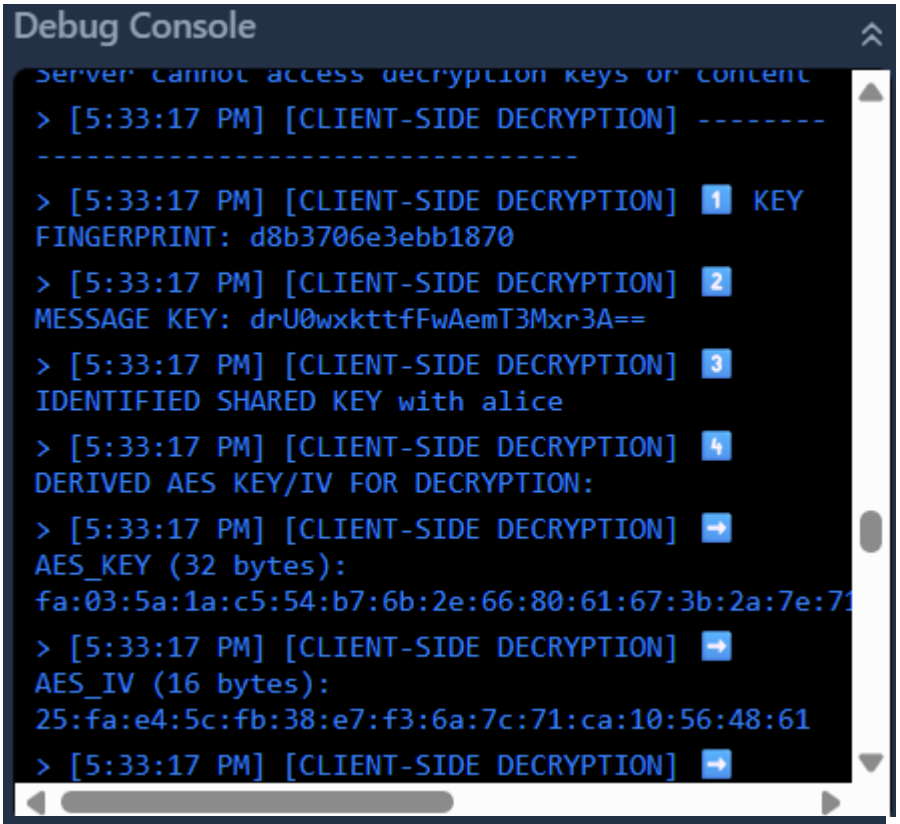
Encryption logs on Alice End:

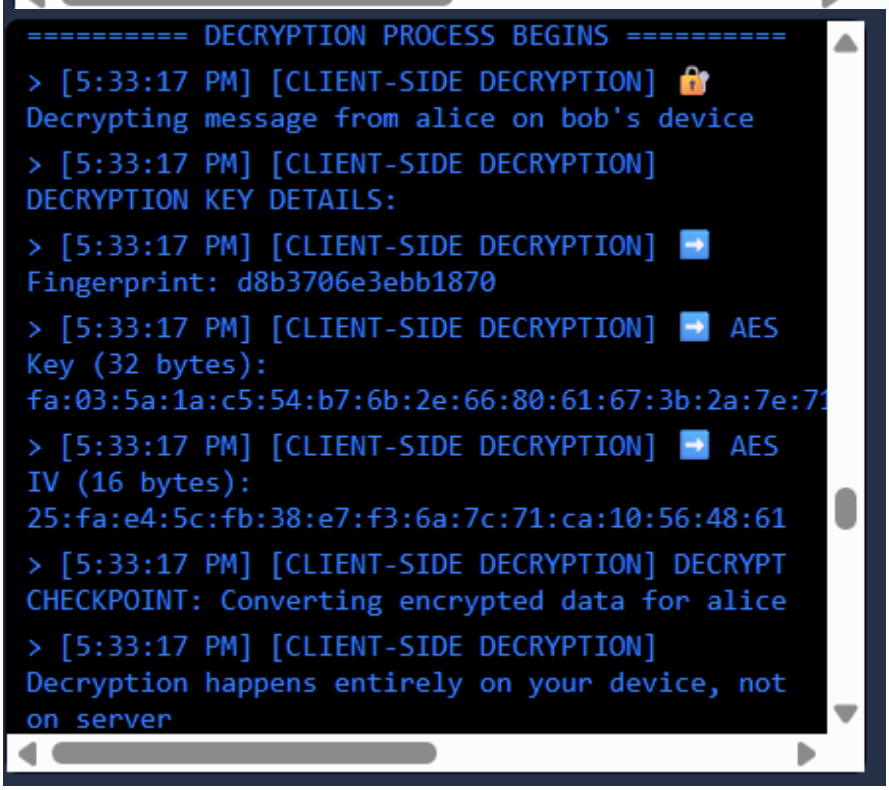


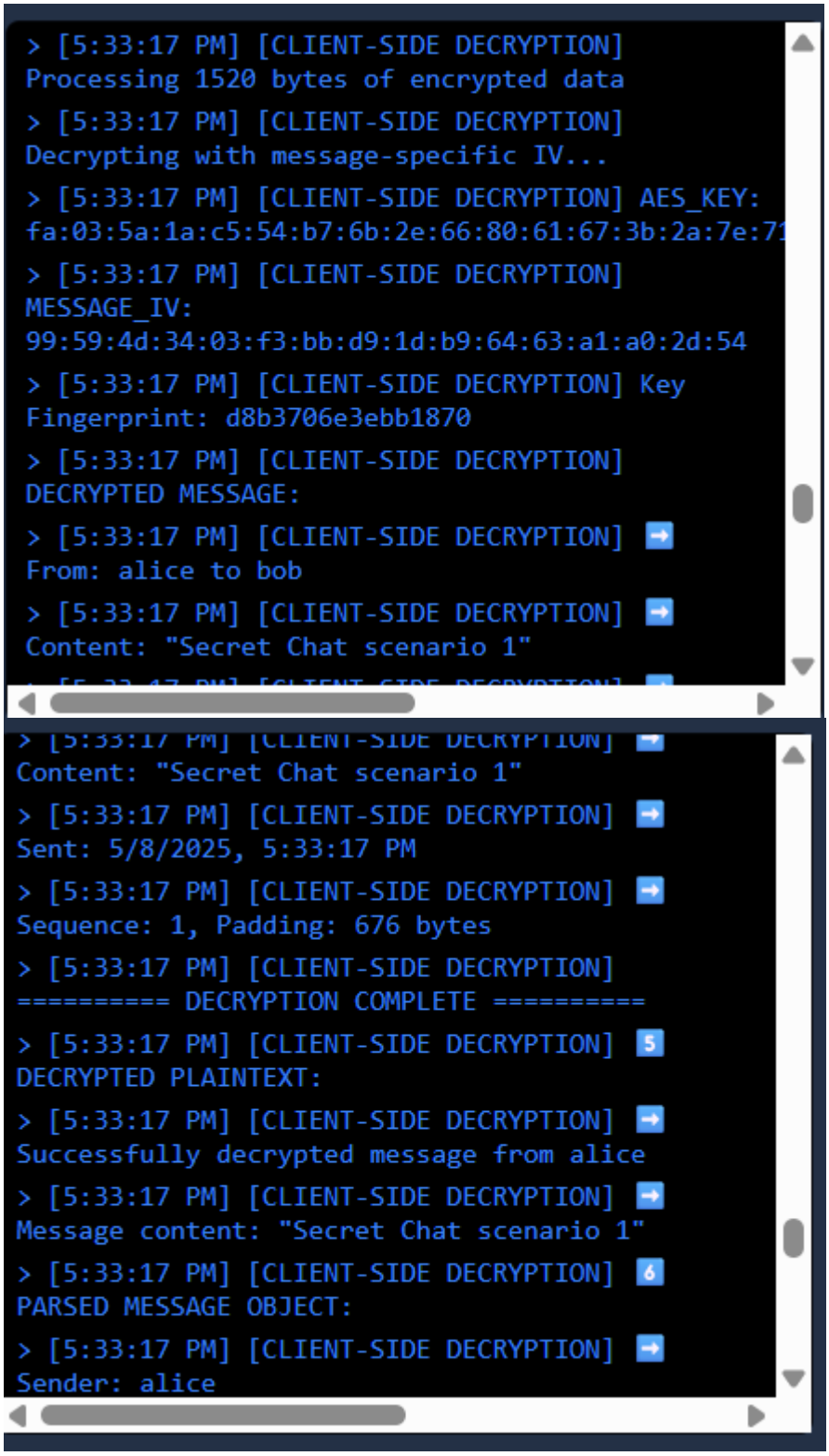


Decryption logs on Bob's end:




Debug Console
Server cannot access decryption keys or content
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] --------
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] 1 KEY FINGERPRINT: d8b3706e3ebb1870
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] 2 MESSAGE KEY: drU0wxkttfFwAemT3Mxr3A==
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] 3 IDENTIFIED SHARED KEY with alice
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] 4 DERIVED AES KEY/IV FOR DECRYPTION:
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] AES_KEY (32 bytes): fa:03:5a:1a:c5:54:b7:6b:2e:66:80:61:67:3b:2a:7e:71
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] AES_IV (16 bytes): 25:fa:e4:5c:fb:38:e7:f3:6a:7c:71:ca:10:56:48:61
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION]

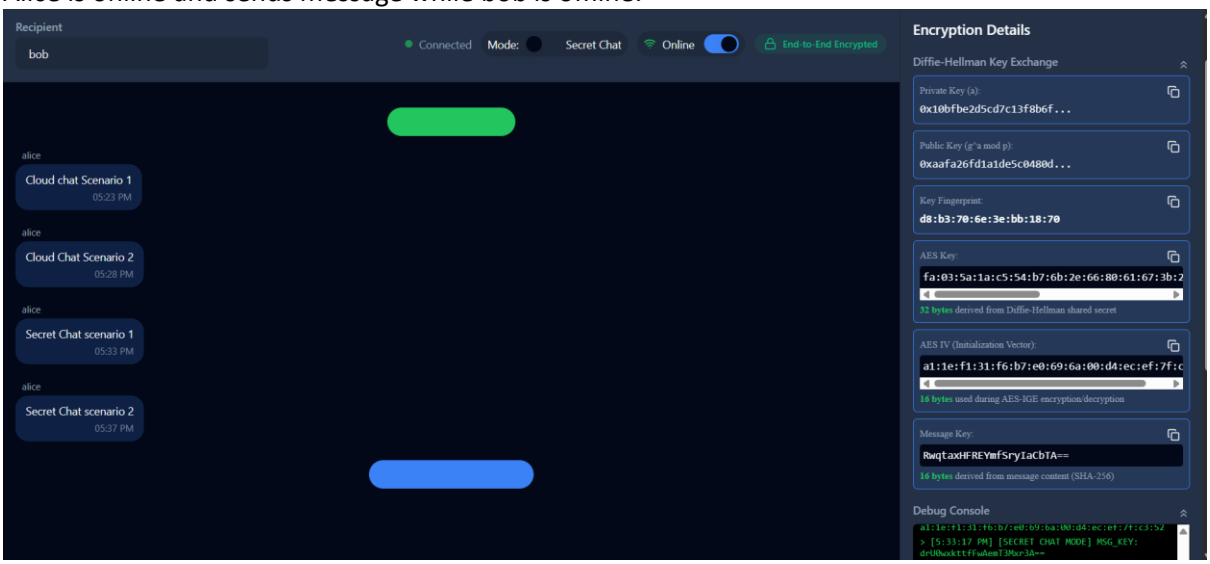

========== DECRYPTION PROCESS BEGINS ==========
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] Decrypting message from alice on bob's device
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] DECRYPTION KEY DETAILS:
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] Fingerprint: d8b3706e3ebb1870
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] AES Key (32 bytes): fa:03:5a:1a:c5:54:b7:6b:2e:66:80:61:67:3b:2a:7e:71
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] AES IV (16 bytes): 25:fa:e4:5c:fb:38:e7:f3:6a:7c:71:ca:10:56:48:61
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] DECRYPT CHECKPOINT: Converting encrypted data for alice
> [5:33:17 PM] [CLIENT-SIDE DECRYPTION] Decryption happens entirely on your device, not on server



Terminal Logs:

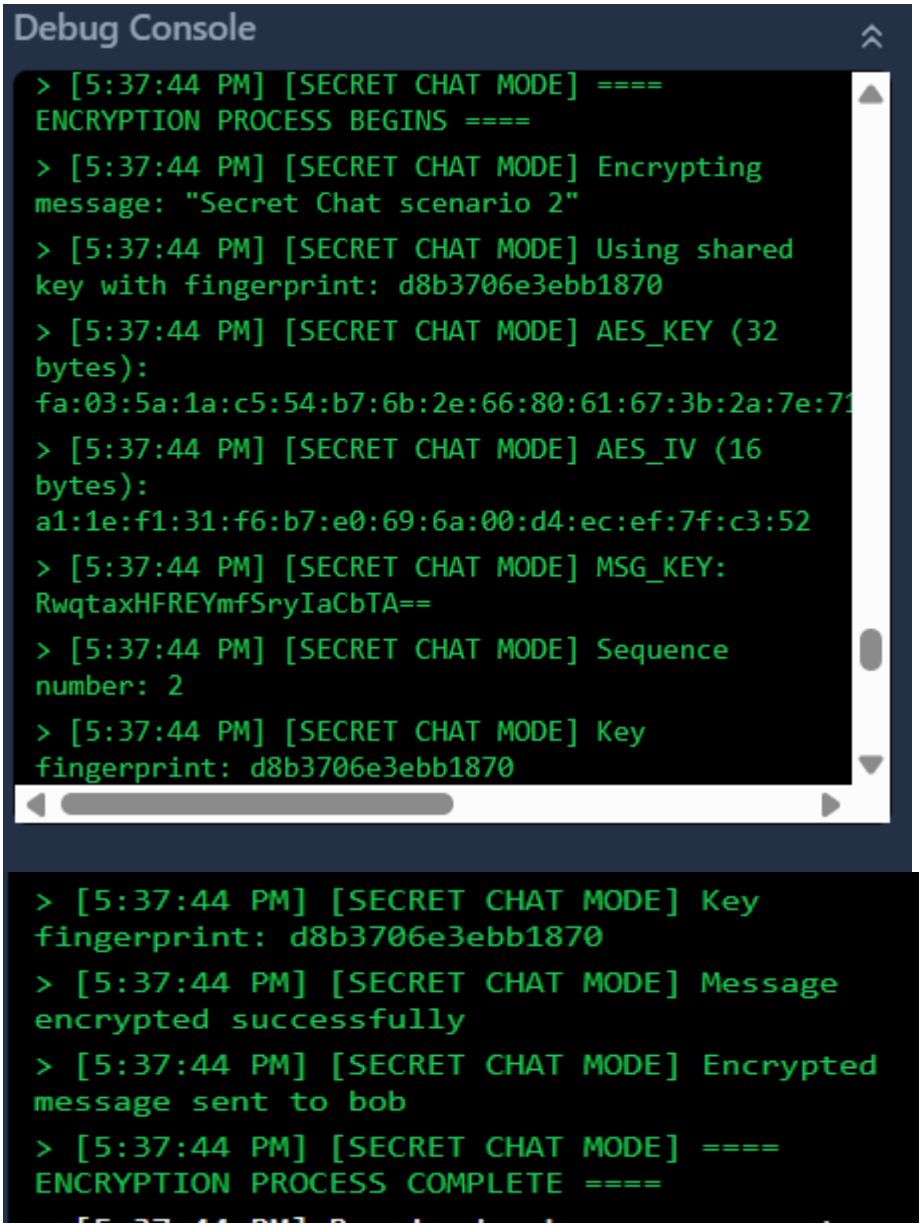
Scenario 2: Alice is Online and Bob is offline

Alice is online and sends message while bob is offline:

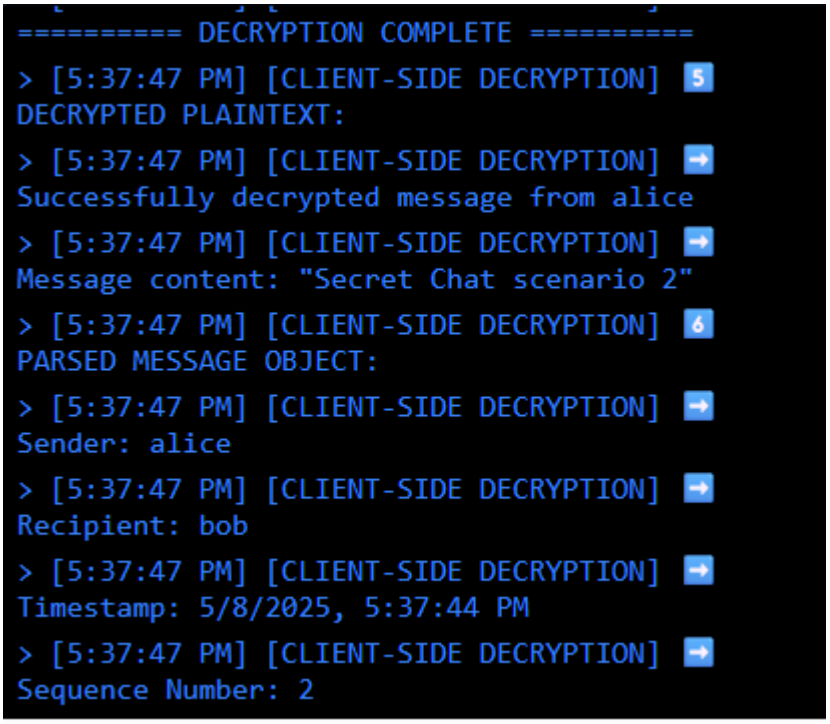


Bob comes online and receives the message:

Encryption logs on Alice end:



Decryption logs on bob's end(For simplicity I have just added the summary logs):

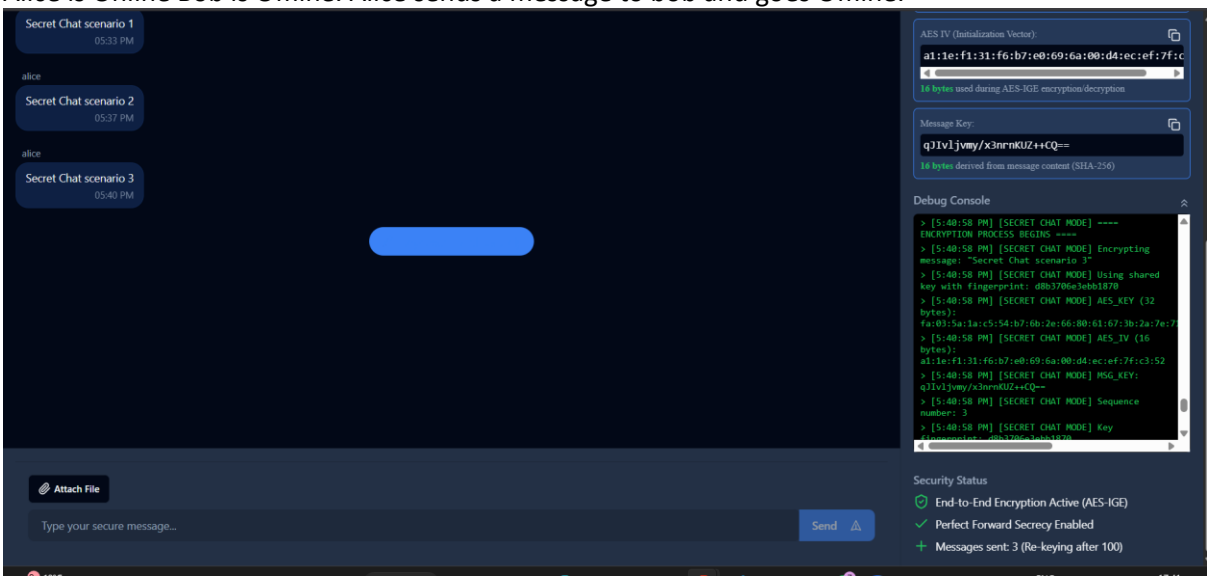

========== DECRYPTION COMPLETE ==========
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION] 5
DECRYPTED PLAINTEXT:
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION]
Successfully decrypted message from alice
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION]
Message content: "Secret Chat scenario 2"
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION] 6
PARSED MESSAGE OBJECT:
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION]
Sender: alice
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION]
Recipient: bob
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION]
Timestamp: 5/8/2025, 5:37:44 PM
> [5:37:47 PM] [CLIENT-SIDE DECRYPTION]
Sequence Number: 2

Terminal Logs:

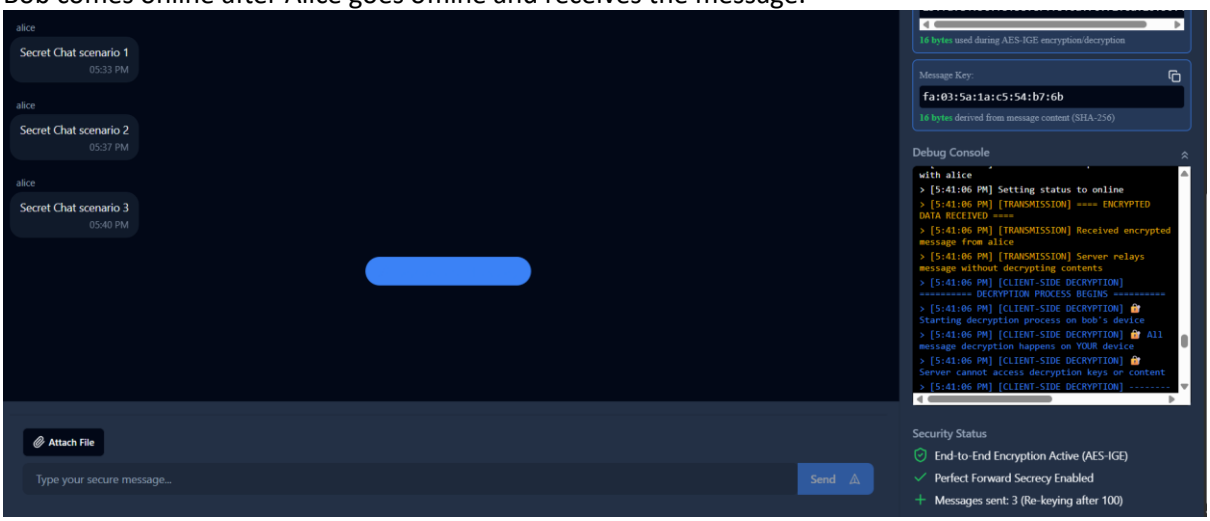

```
Received WebSocket message: {"type":"encrypted-message","recipient":"bob","encryptedMessage":{"msgKey":"RwqtaxHFREYmfSryIaCbTA==","encryptedData":"iJ3tsWUGxIHJ+8k6akd8mWdTQ0tDs6sAN5zOmLgI/yodnhECYm94Ls
JSHEOeDbNmp8B28biVUcO7JPS2pqS/9w4ljfWsD+nFRVsyykLPinlzsj9Tszo0bN1k1gR+Kh0DBwVG5SOhm3ipxCwu9pBPCpzDzVxc/aL2vU1Ar1l5tsFgjjcD387aPKN82oXBImG67R1p6Sp4y7c74IhuVqKzESTbOp+8QRsEXqXE4jDG4eYCj753gUzADbZiJCFiWo5
qAtQ/+0eQmKsy+21K1NQWBVKUREdnbGIif9s2u80fugUJSYdKbWH+K7plMqxoZHM6DtJID0nv5mDn6vIgOUAtAJOGzGnmVjhsEUqvrgmTKz37DFTFbYAGrhr+mclNybJtHPnQmKawl+CtyCcBc3EUMpUhMQpIpngmBLh6/1iVKwU6GBur4sajqyASKO65Um/VdGniufZn
CB09lgWB1QD3I/Xe2p1uEwkehxswUspJgMk=","keyFingerprint":"d8b3706e3ebb1870","sequenceNumber":2,"senderUsername":"alice","iv":"0m+RB8Od6MwVvG9u60tgLw=="},"timestamp":1746743864097,"id":"enc-1746743864097-
bq1jp90"}
Parsed message type: encrypted-message
Processing encrypted message: {
  sender: 'alice',
  recipient: 'bob',
  hasEncryptedMessage: true,
  hasMessageKeys: 'msgKey: true, encData: true'
}
Recipient "bob" online status for message: false
Recipient bob is offline, storing message for later delivery
```


Scenario 3: Alice is Online Bob is Offline. Alice sends a message to bob and goes Offline. Bob comes online after Alice goes offline.

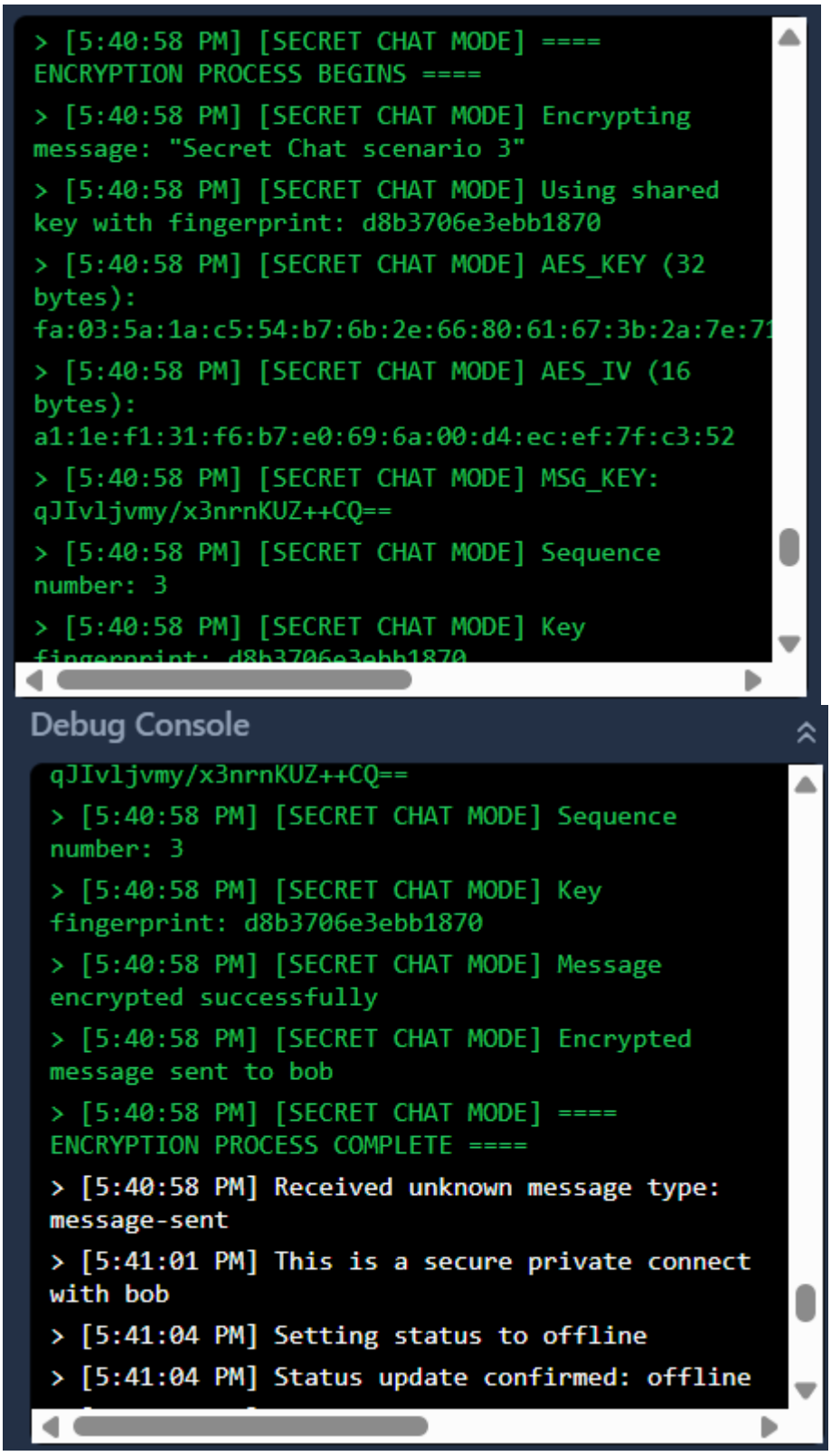
Alice is Online Bob is Offline. Alice sends a message to bob and goes Offline:



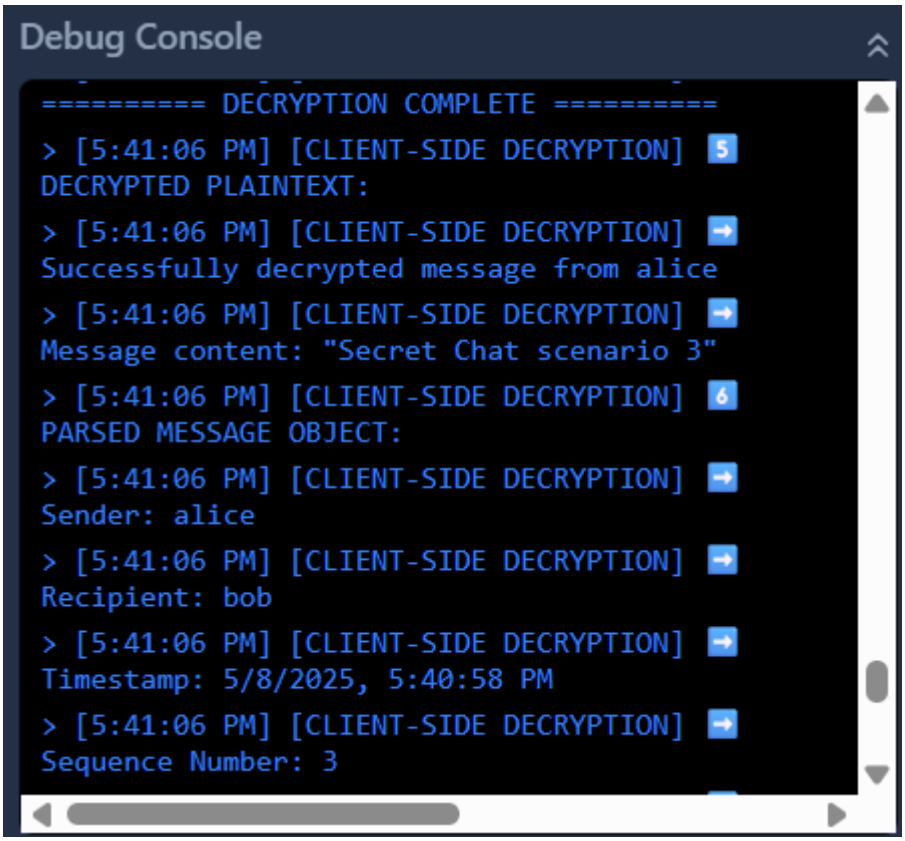
Bob comes online after Alice goes offline and receives the message:



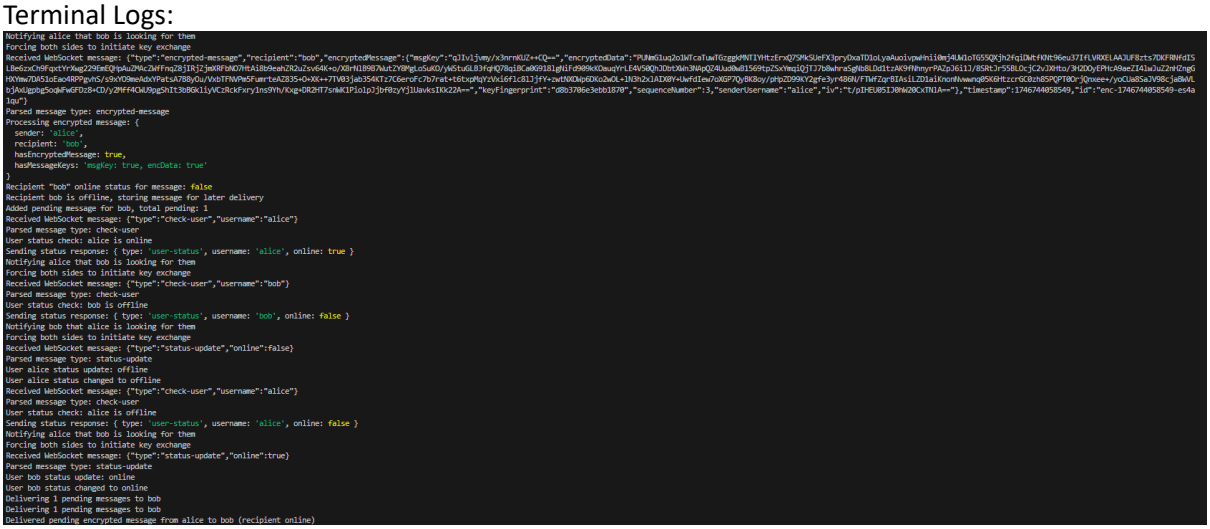
Encryption logs on Alice end:



Decryption logs on Bob's end:



Terminal Logs:



Forward Secrecy:

To enhance long-term security and prevent retroactive message compromise, SecureMessenger implements a key rotation strategy in its Secret Chat mode. Each key is monitored for usage, and once it has been used to encrypt and decrypt more than 100 messages, or has been active for over one week, a re-keying process is done.


Attach File
Type your secure message...
Send
Security Status
End-to-End Encryption Active (AES-IGE)
Perfect Forward Secrecy Enabled
Messages sent: 3 (Re-keying after 100)