Assignment 3 Report

Liam Tarry

July 31st 2025

Student Name: [Liam Tarry]
Student ID: [V00939002]
Assignment: [Assignment 3]
Date: [July 31st]
GitHub Repository: [(https://github.com/ltarry/ECE572_Summer2025_SecureText))]

Executive Summary

[Write your executive summary here]

Table of Contents

1. Introduction
2. Technical Implementation
3. Session Management
4. Security Demonstrations
5. Threat Analysis
6. Perfomance Analysis
7. Conclusion
8. References

---

1. Introduction

1.1 Objective

The main objective of this assignment was to implement end-to-end encryption between users on our python messaging app using elliptic curve diffie-hellman exchange for our public private key exchange and AES-GCM encryption for our symmetric key used between users. A 30 minute session expiration was also implemented to ensure time-limited secrecy and support forward secrecy between users.

1.2 Scope

The primary implementations and key elements I focused on were properly implementing ECDH and AES encryption between users along with a functioning session management element on both the server and client side. The most important elements focused on were the security principles necessary that come with the hybrid encryption and proper session management and cleanup including forward-secrecy, session security, confidentiality, integrity, and authentication.

1.3 Environment Setup

- **Operating System**: Kali Linux
- **Python Version**: 3.13.5
- **Key Libraries Used**: Cryptography (specifically: HKDF, AESGCM, invalidtag) and prior used in assignments 2 and 1. See head of python file for full list.
- **Development Tools**: VSCode, Github Co-pilot

---

2 & 3. Technical Implementation - ECDH and AES-GCM Design & Session Management Design Implementation

Code Snippet (Key Implementations):

```
# Include only the most important code snippets
# Do not paste entire files as the actual attack or security-fixed codes are
↪  included in the deliverables directory

#Required Imports for the Technical Implementation:
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.exceptions import InvalidTag

#Added to our SecureTextClientInit for E2EE and Session Management:
        # For end-to-end encryption
        self.private_key = None
        self.public_key = None
        self.peer_keys = {}  # username -> public key

        # Session management
        self.last_activity = time.time()
        self.session_timeout = 30 * 60  # 30 minutes
        self.warning_threshold = 25 * 60  # 25 minutes (5-min warning)

#Across authentication methods added these to reset activity
current_user = None  # Reset the user state
last_activity = time.time()  # Reset the activity timer
action_count = 0  # Reset the action counter
```

```
#Added the following methods to our SecureTextClient class for ECDH Key
↪  Management:
def generate_keys(self):
    """Generate ECDH key pair using P-256 curve"""
    self.private_key = ec.generate_private_key(ec.SECP256R1())
    self.public_key = self.private_key.public_key()
    print("[+] Generated new ECDH key pair")

def serialize_public_key(self):
    """Serialize public key for transmission"""
    if not self.public_key:
        return None

    public_bytes = self.public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    return base64.b64encode(public_bytes).decode('utf-8')

def deserialize_public_key(self, public_key_b64):
    """Deserialize received public key"""
    if not public_key_b64:
        return None

    try:
        public_bytes = base64.b64decode(public_key_b64)
        peer_public_key = serialization.load_pem_public_key(public_bytes)
        return peer_public_key
    except Exception as e:
        print(f"Error deserializing public key: {e}")
        return None

def derive_shared_key(self, other_user):
    """Derive shared key using ECDH and HKDF-SHA256"""
    if not self.private_key or other_user not in self.peer_keys:
        return None

    try:
        # Normalize self username if it's a GitHub user
        normalized_self = self.username
        if self.oauth_logged_in and '_' in self.username:
            normalized_self = self.username.split('_')[0]

        # Normalize other username if it has an underscore (GitHub user)
        normalized_other = other_user
```

```
        if '_' in other_user:
            normalized_other = other_user.split('_')[0]

        # Get other user's public key
        peer_public_key = self.peer_keys[other_user]

        # Perform ECDH key exchange
        shared_secret = self.private_key.exchange(ec.ECDH(), peer_public_key)

        # Use normalized usernames for key derivation
        usernames = sorted([normalized_self, normalized_other])
        info = f"SecureText-{usernames[0]}-{usernames[1]}".encode('utf-8')

        print(f"[DEBUG] Using info string for key derivation: '{info.decode()}'")

        derived_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,  # 256 bits for AES-256
            salt=None,
            info=info
        ).derive(shared_secret)
        print(f"[CRYPTO] Derived {len(derived_key)*8}-bit AES key using ECDH with
        ↪ {other_user}")
        return derived_key
    except Exception as e:
        print(f"Error deriving shared key: {e}")
        return None

#Added the following methods to SecureTexClient class as well for AES-GCM Message
↪ Encryption:
def encrypt_message(self, recipient, plaintext):
        """Encrypt message using AES-256-GCM"""
        # Update activity timestamp
        self.update_activity()

        # Derive key for this recipient
        key = self.derive_shared_key(recipient)
        if not key:
            return None, None

        try:
            # Create AES-GCM cipher
            aesgcm = AESGCM(key)

            # Generate a unique 96-bit nonce for each message
            nonce = os.urandom(12)
            print(f"[CRYPTO] Generated unique {len(nonce)*8}-bit nonce for this
            ↪ message")
```

```
            # Encrypt with associated data (AD) for integrity
            # AD contains sender and recipient to prevent replay attacks
            normalized_sender = self.username
            if self.oauth_logged_in:
                # Use just the GitHub username part without suffix
                normalized_sender = normalized_sender.split('_')[0]
            ad = f"{normalized_sender}:{recipient}".encode('utf-8')

            # Encrypt the plaintext
            ciphertext = aesgcm.encrypt(nonce, plaintext.encode('utf-8'), ad)
            print(f"[CRYPTO] Message encrypted with AES-256-GCM and authenticated
            ↪  with sender-recipient IDs")
            print(f"[DEBUG] Using AD for encryption:
            ↪  '{normalized_sender}:{recipient}'")
            # Return base64 encoded nonce and ciphertext
            return base64.b64encode(nonce).decode('utf-8'),
            ↪  base64.b64encode(ciphertext).decode('utf-8')
        except Exception as e:
            print(f"Encryption error: {e}")
            return None, None

def decrypt_message(self, sender, nonce_b64, ciphertext_b64):
    """Decrypt message using AES-256-GCM"""
    #For key exchange display/ debug
    print(f"[DEBUG] Attempting to decrypt message from {sender}")
    print(f"[DEBUG] Do I have {sender}'s key? {'Yes' if sender in self.peer_keys
    ↪  else 'No'}")
    # Update activity timestamp
    self.update_activity()

    # Normalize sender name if it's a GitHub user
    normalized_sender = sender
    if '_' in sender:  # Likely a GitHub user
        normalized_sender = sender.split('_')[0]

    # Derive key for this sender
    key = self.derive_shared_key(sender)
    if not key:
        return None

    try:
        # Decode nonce and ciphertext from base64
        nonce = base64.b64decode(nonce_b64)
        ciphertext = base64.b64decode(ciphertext_b64)

        # Create AES-GCM cipher
        aesgcm = AESGCM(key)
```

```
            # Use the normalized sender name for AD
            ad = f"{normalized_sender}:{self.username}".encode('utf-8')
            print(f"[DEBUG] Using AD for decryption:
            ↪  '{normalized_sender}:{self.username}'")

            # Decrypt the ciphertext
            plaintext = aesgcm.decrypt(nonce, ciphertext, ad)

            #Integrity check for demonstration purposes
            print(f"[SECURITY] Message integrity verified using AES-GCM
            ↪  authentication")
            return plaintext.decode('utf-8')
        except InvalidTag:
            print(f"[SECURITY] INTEGRITY VIOLATION! Message has been tampered with or
            ↪  corrupted.")
            return None
        except Exception as e:
            print(f"Decryption error: {e}")
            return None

#With the next step below, I added the following methods for session management
↪  (30 minute expiry) tou our SecureTextClient Class:
def update_activity(self):
    """Update last activity timestamp"""
    self.last_activity = time.time()
    remaining = self.session_timeout - (time.time() - self.last_activity)
    print(f"[SESSION] Activity detected! Session valid for {int(remaining/60)}m
    ↪  {int(remaining%60)}s")

def check_session(self):
    """Check if session is about to expire or has expired"""
    if not self.logged_in:
        return True

    now = time.time()
    elapsed = now - self.last_activity

    # Session expired
    if elapsed > self.session_timeout:
        print("\n[!] Session expired. Please log in again.")
        self.logout(expired=True)
        return False

    # Warning before expiration
    if elapsed > self.warning_threshold:
        remaining = self.session_timeout - elapsed
        print(f"\n[!] Warning: Session will expire in {int(remaining/60)} minutes
        ↪  and {int(remaining%60)} seconds.")
```

```
    return True

def monitor_session(self):
    """Periodically monitor session status"""
    while self.running:
        self.check_session()
        time.sleep(60)  # Check every minute

def logout(self, expired=False):
    """Securely logout and clean up sensitive material"""
    # Secure cleanup of cryptographic material
    self.private_key = None
    self.public_key = None
    self.peer_keys.clear()

    # Reset session state
    self.logged_in = False
    self.running = False
    self.username = None
    self.oauth_logged_in = False
    self.access_token = None

    if not expired:
        print("Logged out successfully")

    # Force garbage collection to clear sensitive data from memory
    import gc
    gc.collect()

#The next step was adding to our SecureTextServer class, in our handle_client
↪  method, the additional command handling for server-side public key storage:
elif command == 'STORE_PUBLIC_KEY':
        if not current_user:
            response = {'status': 'error', 'message': 'Not logged in'}
            self.log_event(
                event="STORE_PUBLIC_KEY",
                username=None,
                outcome="DENIED",
                details="Attempted to store public key while not logged in"
            )
        else:
            public_key = message.get('public_key')
            if not public_key:
                response = {'status': 'error', 'message': 'No public key
↪  provided'}
            else:
                # Store the public key in the user's record
```

```
                self.users[current_user]['public_key'] = public_key
                self.save_users()
                response = {'status': 'success', 'message': 'Public key stored'}
                self.log_event(
                    event="STORE_PUBLIC_KEY",
                    username=current_user,
                    outcome="SUCCESS",
                    details="Public key stored"
                )

elif command == 'GET_PUBLIC_KEY':
        if not current_user:
            response = {'status': 'error', 'message': 'Not logged in'}
            self.log_event(
                event="GET_PUBLIC_KEY",
                username=None,
                outcome="DENIED",
                details="Attempted to get public key while not logged in"
            )
        else:
            target_user = message.get('username')
            if not target_user or target_user not in self.users:
                response = {'status': 'error', 'message': 'User not found'}
            elif 'public_key' not in self.users[target_user]:
                response = {'status': 'error', 'message': 'User has no public
↪  key'}
            else:
                response = {
                    'status': 'success',
                    'username': target_user,
                    'public_key': self.users[target_user]['public_key']
                }
                self.log_event(
                    event="GET_PUBLIC_KEY",
                    username=current_user,
                    outcome="SUCCESS",
                    details=f"Retrieved {target_user}'s public key"
                )

#After this step, I then added the following additions to the login(self) method
↪  in the SecureTextClient to start monitoring the session, generate ECDH keys,
↪  and store the public keys on the server on a successful login:
if response['status'] == 'success':
        self.logged_in = True
        self.username = username
        self.role = response.get('role', 'user')
        self.running = True
        self.update_activity()  # Initialize activity tracking
```

```
        # Generate ECDH keys for E2EE
        self.generate_keys()

        # Store public key on server
        pub_key = self.serialize_public_key()
        command = {
            'command': 'STORE_PUBLIC_KEY',
            'public_key': pub_key
        }
        response = self.send_command(command)
        if response['status'] != 'success':
            print(f"Warning: {response['message']}")

        # Start listening for messages
        listen_thread = threading.Thread(target=self.listen_for_messages)
        listen_thread.daemon = True
        listen_thread.start()

        # Start session monitoring
        session_thread = threading.Thread(target=self.monitor_session)
        session_thread.daemon = True
        session_thread.start()

#Additionally, added key generation to a challenge response login and github
↪  login for consistency across methods with the following code added after
↪  running=true on a successful login:

    # Generate ECDH keys for E2EE
    self.generate_keys()

    # Store public key on server
    pub_key = self.serialize_public_key()
    command = {
        'command': 'STORE_PUBLIC_KEY',
        'public_key': pub_key
    }
    response = self.send_command(command)
    if response['status'] != 'success':
        print(f"Warning: {response['message']}")

    # Start session monitoring
    session_thread = threading.Thread(target=self.monitor_session)
    session_thread.daemon = True
    session_thread.start()

#The following code now shows my implementation for the E2EE in the
↪  send_message(self) method in SecureTextClient:
```

```
def send_message(self):
    """Send an encrypted message to another user"""
    if not self.logged_in:
        print("You must be logged in to send messages!")
        return

    # Check session before continuing
    if not self.check_session():
        return

    self.update_activity()

    print("\n=== Send Encrypted Message ===")
    recipient = input("Enter recipient username: ").strip()
    # Check if we're trying to message a GitHub user
        if recipient not in all_users:
            # Look for username with suffix
            for username in all_users:
                if username.startswith(recipient + '_'):
                    print(f"Note: Using {username} instead of {recipient} (GitHub
                    ↪  account)")
                    recipient = username
                    break
    content = input("Enter message: ").strip()

    if not recipient or not content:
        print("Recipient and message cannot be empty!")
        return

    # Get recipient's public key if we don't have it
    if recipient not in self.peer_keys:
            command = {
                'command': 'GET_PUBLIC_KEY',
                'username': recipient
            }
            print(f"[DEBUG] Fetching {recipient}'s public key for encryption...")
            response = self.send_command(command)
            if response['status'] != 'success':
                print(f"Error: {response['message']}")
                return

            recipient_public_key =
↪   self.deserialize_public_key(response['public_key'])
            if not recipient_public_key:
                print("Error: Invalid public key for recipient")
                return

            self.peer_keys[recipient] = recipient_public_key
```

```
            print(f"[DEBUG] Successfully retrieved {recipient}'s public key")

        # Encrypt the message
    print(f"[DEBUG] Encrypting message with AES-256-GCM...")
    nonce_b64, ciphertext_b64 = self.encrypt_message(recipient, content)
    if not nonce_b64 or not ciphertext_b64:
        print("Error: Could not encrypt message")
        return

    # Send the encrypted message
    command = {
        'command': 'SEND_MESSAGE',
        'recipient': recipient,
        'nonce': nonce_b64,
        'ciphertext': ciphertext_b64
    }

    response = self.send_command(command)
    print(f"{response['message']}")

#Having now updated the send_message method, I needed to update the handler in
↪  our SecureTextServer handle_client method with the following:
elif command == 'SEND_MESSAGE':
    if not current_user:
        response = {'status': 'error', 'message': 'Not logged in'}
        self.log_event(
            event="SEND_MESSAGE",
            username=None,
            outcome="DENIED",
            details="Attempted to send message while not logged in"
        )
    else:
        recipient = message.get('recipient')
        # For E2EE, we now receive encrypted message components
        nonce = message.get('nonce')  # Base64 encoded
        ciphertext = message.get('ciphertext')  # Base64 encoded

        # Validate we have all required fields
        if not recipient or not nonce or not ciphertext:
            response = {'status': 'error', 'message': 'Missing required fields'}
        else:
            # Send message to recipient if they're online
            if recipient in self.active_connections:
                msg_data = {
                    'type': 'MESSAGE',
                    'from': current_user,
                    'nonce': nonce,
                    'ciphertext': ciphertext,
```

```
                    'timestamp': datetime.now().isoformat()
                }
                try:
                    self.active_connections[recipient].send(
                        json.dumps(msg_data).encode('utf-8')
                    )
                    response = {'status': 'success', 'message': 'Message sent'}
                    self.log_event(
                        event="SEND_MESSAGE",
                        username=current_user,
                        outcome="SUCCESS",
                        details=f"Message sent to {recipient}"
                    )
                except:
                    # Remove inactive connection
                    del self.active_connections[recipient]
                    response = {'status': 'error', 'message': 'Recipient is
↪  offline'}
            else:
                response = {'status': 'error', 'message': 'Recipient is offline'}
    print(f"[SERVER] Relaying encrypted message from {current_user} to
    ↪  {recipient}")
    print(f"[SERVER] Message content (encrypted): nonce={nonce[:10]}...,
    ↪  ciphertext={ciphertext[:20]}...")
    print(f"[SERVER] Server CANNOT decrypt this message - no access to keys")


    self.log_encrypted_message(current_user, recipient, nonce, ciphertext)

#Added for logging purposes at the end of my SecureTextServer Class:
def log_encrypted_message(self, sender, recipient, nonce, ciphertext):
    """Log encrypted messages to a file for demonstration"""
    with open('encrypted_messages.log', 'a') as f:
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        f.write(f"[{timestamp}] FROM:{sender} TO:{recipient}\n")
        f.write(f"NONCE: {nonce[:10]}...\n")
        f.write(f"CIPHERTEXT: {ciphertext[:30]}...\n\n")

#Finally, I neded to update the listen_for_messages method call to account for
↪  the encrypted messages handling:
def listen_for_messages(self):
    """Listen for incoming messages in a separate thread"""
    while self.running:
        try:
            data = self.socket.recv(1024).decode('utf-8')
            if data:
                message = json.loads(data)
                # Check for session timeout
```

```
                if message.get('status') == 'error' and "timed out" in
                ↪  message.get('message', '').lower():
                    print(f"\n[!] Session timed out. Reason:
                    ↪  {message.get('message')} Logging out.")
                    self.logout()
                    break

                if message.get('type') == 'MESSAGE':
                    sender = message['from']
                    self.update_activity()  # Update activity on receiving
                    ↪  message

                    # Handle encrypted messages
                    if 'nonce' in message and 'ciphertext' in message:
                        # Get sender's public key if we don't have it
                        if sender not in self.peer_keys:
                            command = {
                                'command': 'GET_PUBLIC_KEY',
                                'username': sender
                            }
                            response = self.send_command(command)
                            if response['status'] != 'success':
                                print(f"\nError receiving message:
                                ↪  {response['message']}")
                                continue

                            sender_public_key =
↪  self.deserialize_public_key(response['public_key'])
                            if not sender_public_key:
                                print("\nError: Invalid public key for sender")
                                continue

                            self.peer_keys[sender] = sender_public_key

                        # Decrypt the message
                        nonce_b64 = message['nonce']
                        ciphertext_b64 = message['ciphertext']
                        plaintext = self.decrypt_message(sender, nonce_b64,
↪  ciphertext_b64)

                        if plaintext:
                            print(f"\n[{message['timestamp']}] {sender}:
                            ↪  {plaintext}")
                            print(">> ", end="", flush=True)
                        else:
                            print(f"\nReceived encrypted message from {sender},
                            ↪  but could not decrypt it.")
                            print(">> ", end="", flush=True)
```

```
                    else:
                        # Legacy plaintext messages
                        print(f"\n[{message['timestamp']}] {sender}:
                        ↪  {message['content']}")
                        print(">> ", end="", flush=True)
            except Exception as e:
                if self.running:  # Only log errors if we're supposed to be running
                    print(f"\nError listening for messages: {e}")
                    break

    #Adapted send_command method for debug purposes:
    def send_command(self, command_data):
        """Send command to server and get response"""
        max_retries = 3
        for attempt in range(max_retries):
            try:
                self.socket.settimeout(5.0)  # Set 5-second timeout
                self.socket.send(json.dumps(command_data).encode('utf-8'))
                response = self.socket.recv(1024).decode('utf-8')
                self.socket.settimeout(None)  # Reset timeout
                return json.loads(response)
            except Exception as e:
                print(f"Communication error (attempt {attempt+1}/{max_retries}):
                ↪  {e}")
                time.sleep(1)  # Wait before retry
```

2 & 3.1 Challenges and Solutions The biggest problems I ran into were involved with session management. Due to reset conflicts and forced re-authentication based on connection pre-established, it caused conflicts when a user was logged out forcibly by the server after idle time was met and then. While I was able to refresh the session on login for a standard TOTP login with a user after being logged out, for both challenge-response and GitHub login, these users MUST refresh the entire connection session on a clean re-connect to login again. While this properly enforces the security implementation, it causes reduced convenience and performance for a user essentially having to restart the application entirely to login again if they chose the latter authentication methods. I was unable to make a solution that was able to be properly implemented to allow for those two authentication methods to re-authenticate in a single connection, only a standard login functioned correctly.

2 & 3.2 Testing and Validation The testing and validation for message integrity, tampering, proper key exchange, and session management involved printing the entire handshake process, logging the messages to display their envryption is correctly functioning, attempting to send a message on a new ECDH key to force an integrity violation based on a new session establishment, and waiting for a user to be locked out of their account by leaving the application inactive for 5 minute demo sessions. All testing and validation showed in the following screenshots and video demo.

2 & 3.3 Provider Configuration For OAuth Login: Before starting the server, in your the securetext.py file you run: replace “GITHUB_CLIENT_ID” to “Ov23liKAGQ5DVCq4cMlJ” replace

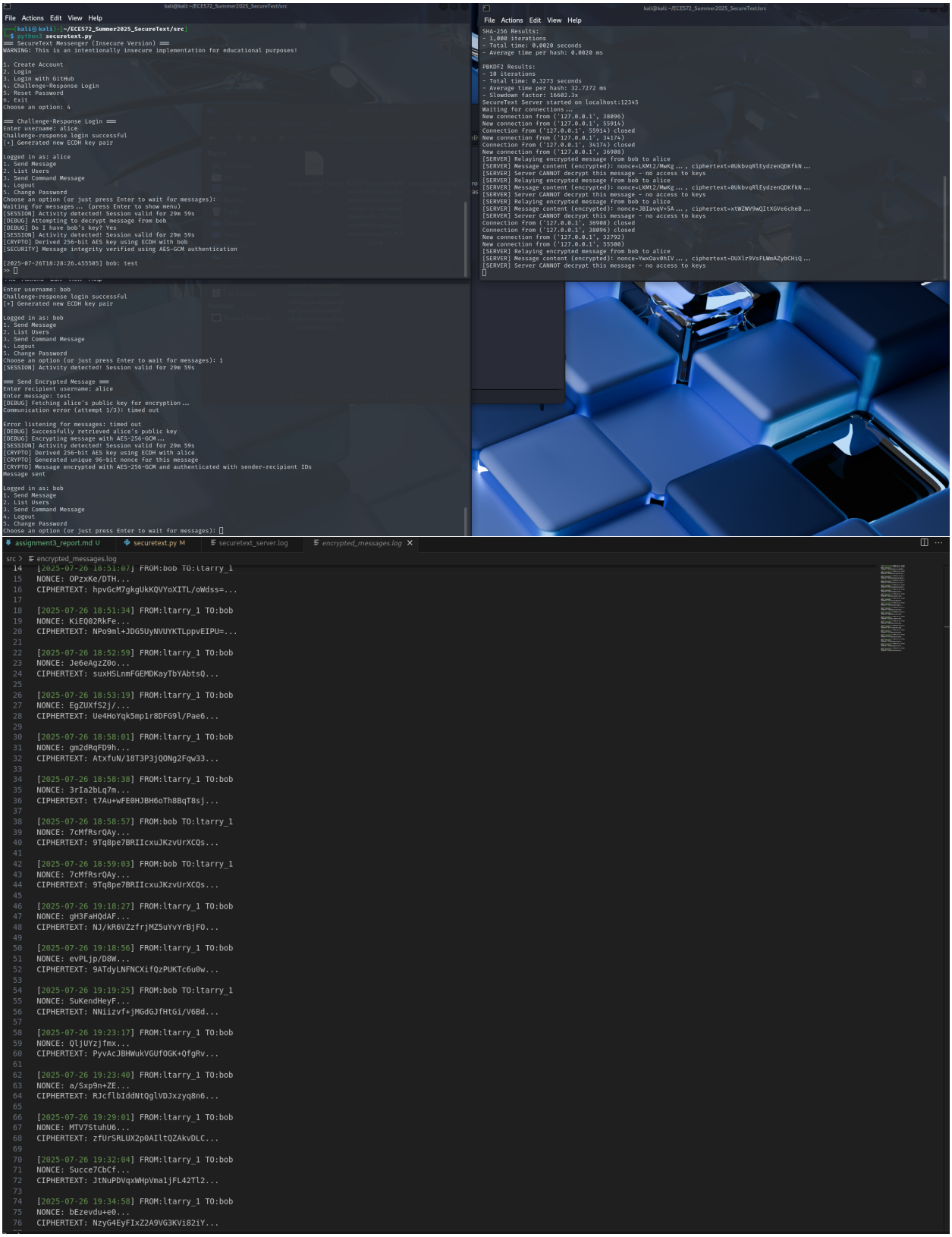
"GITHUB_CLIENT_SECRET" to the client secret submitted on brightspace

Now, when starting server, select option 3 to prompt a firefox browser pop-up, a URL is given if pop-up does not occur. Login with github account credentials and paste redirect URI to console. Once shown in assignment test, best practice is to logout from github to clear cache and nano users.json to remove username/ email from file.

---

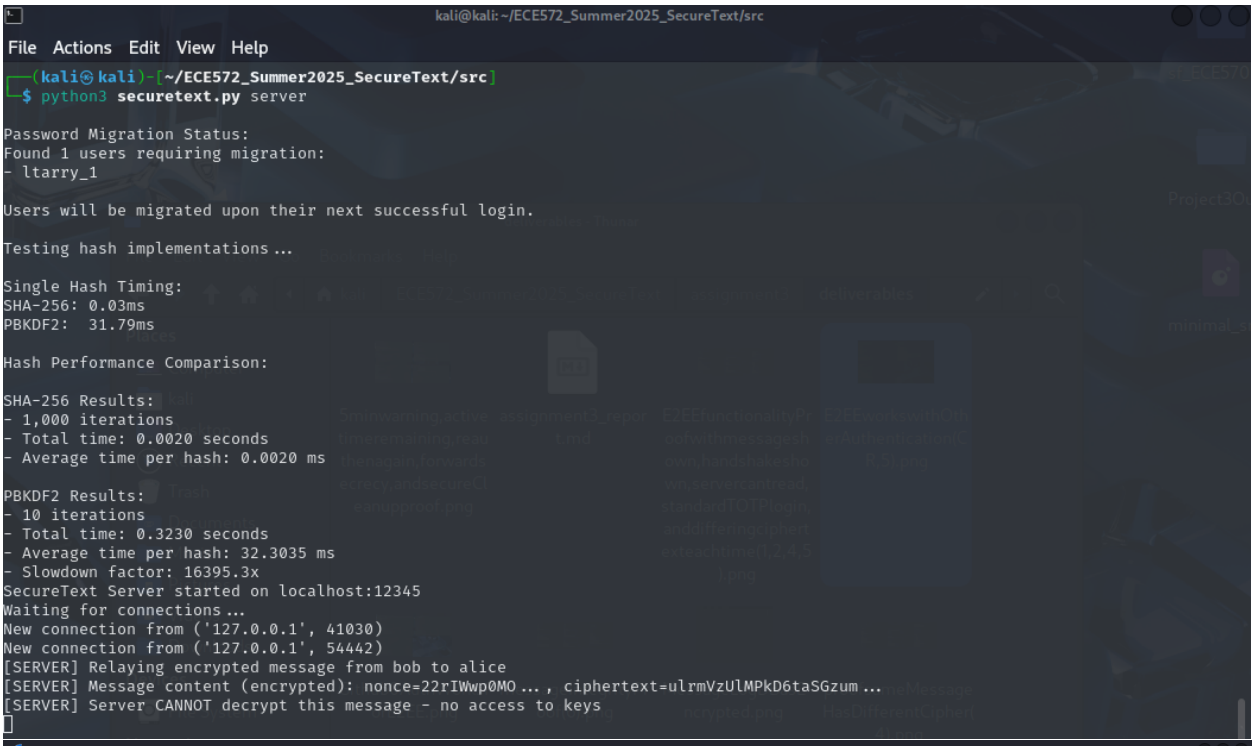
4. Security Demonstrations

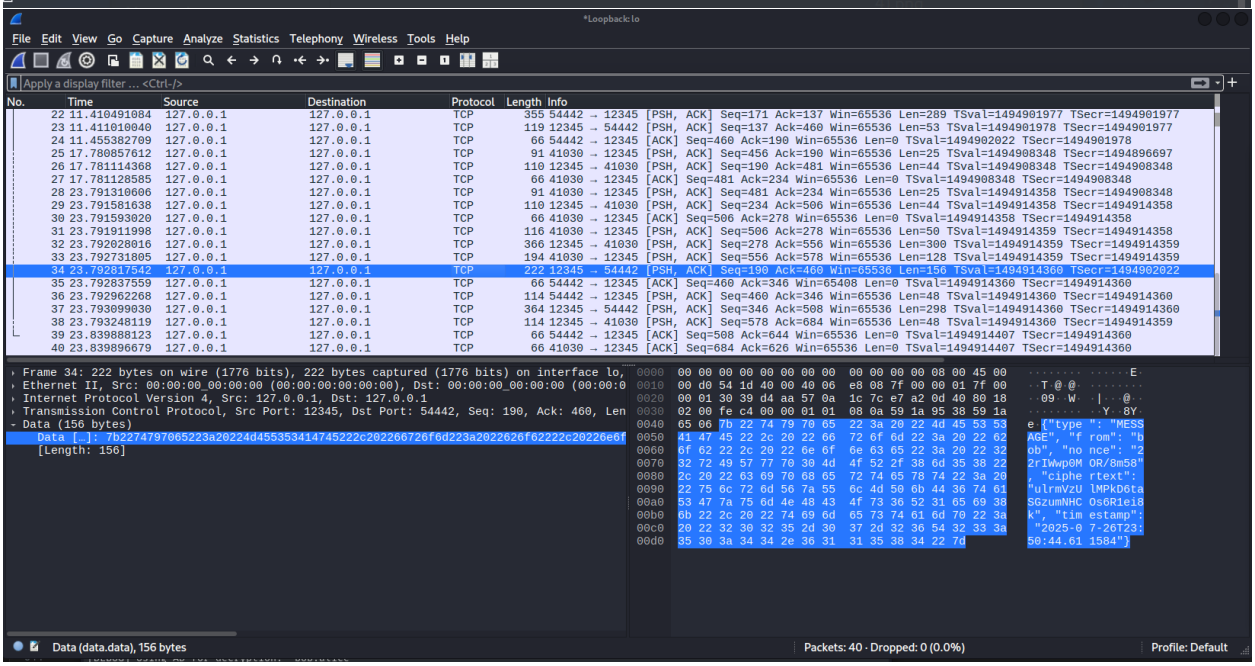
1. E2EE Functionality Proof



In the above two images we first see our E2EE working, the handshake printed to console with details on both client sides for the interaction, along with the server outputting an attempt at accessing data but it cannot, and instead only relays. The second screenshot shows our message databse with encrypted messages stored between users.

2. Server Cannot Decrypt Proof

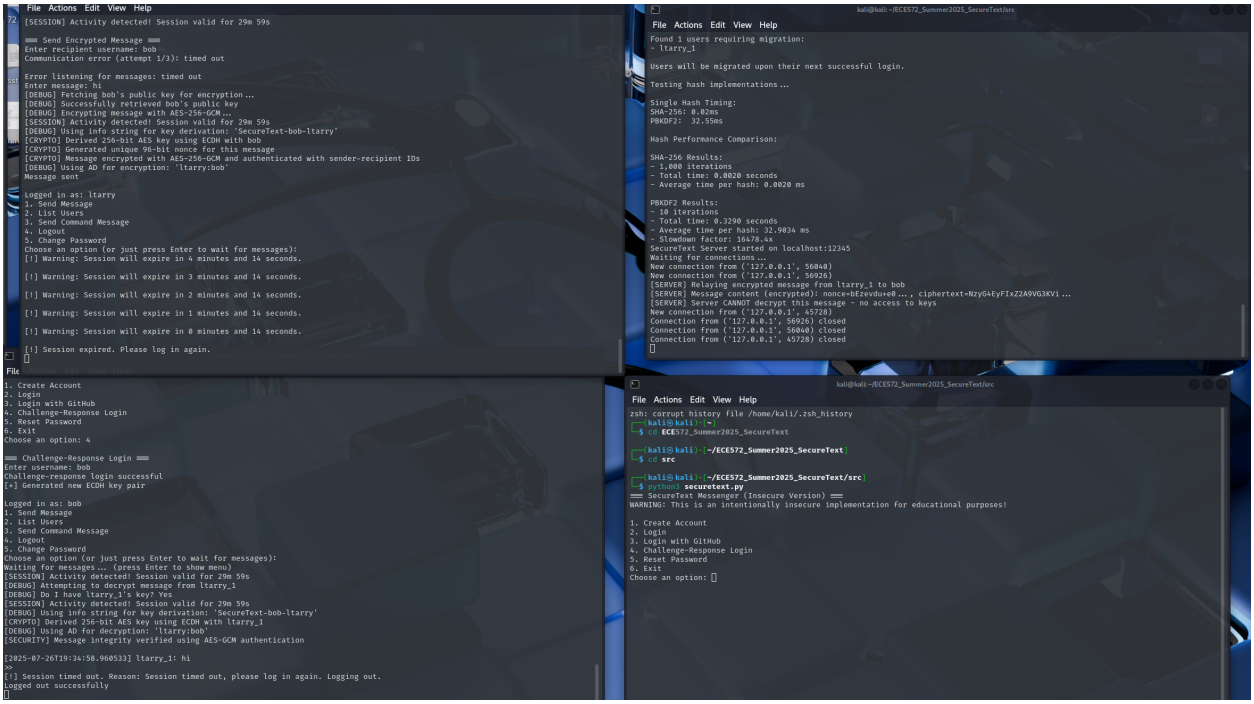




See images in 1 for further proof along with database. In this first image is a screenshot of the server console output proving it cannot decrypt, along with it only routing data. Wireshark screenshot

also shows that in the communicated messages itself only encrypted data is transferred.

3. Session Management Proof

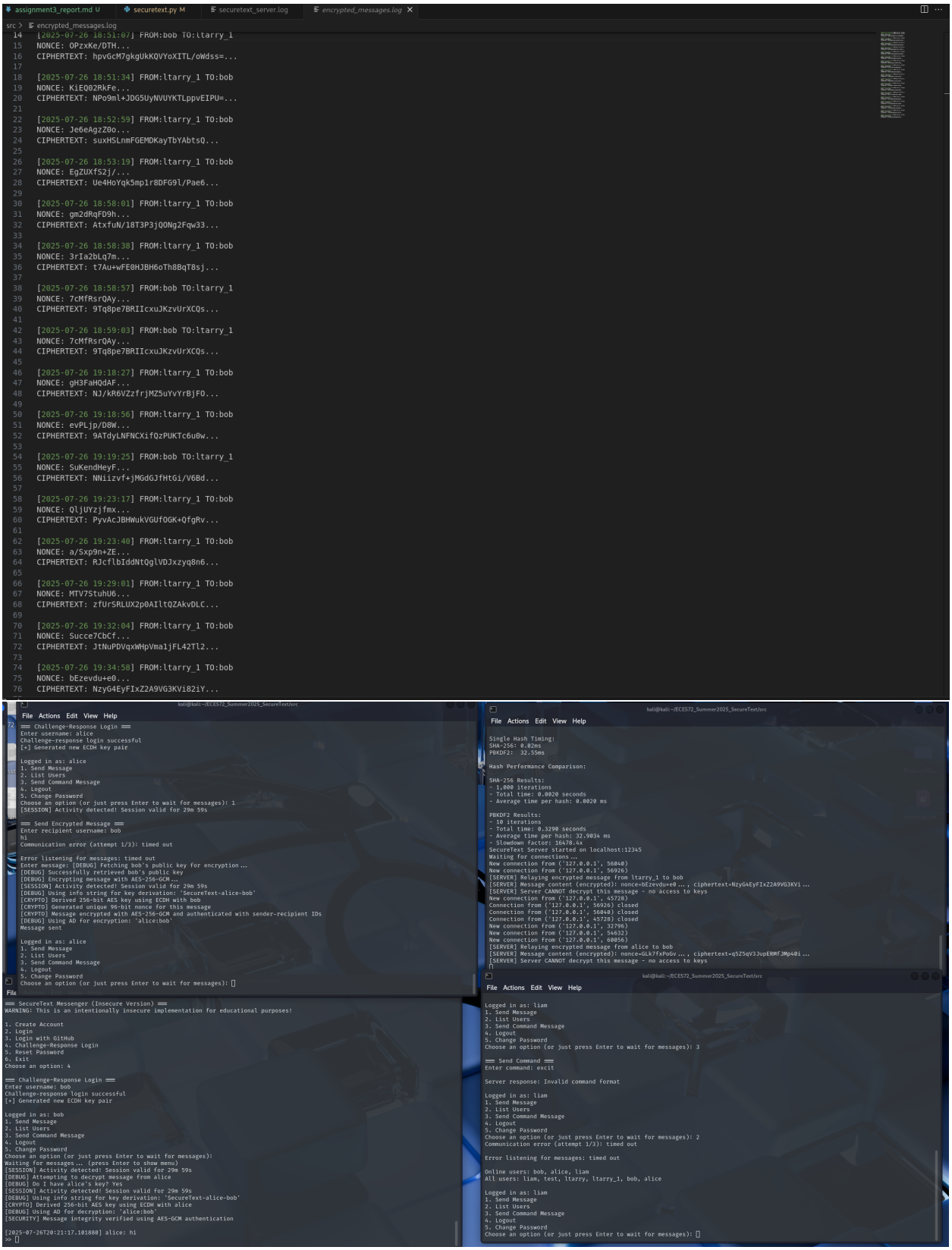


The above screenshot shows: an active session with time remaining (countdown and server proof of message sent), warning countdowns after only 5 minutes remain in the live session, a forced logout and asking the user to login again, along with a session cleanup on the server terminal where the connections are properly closed.

4. Cryptographic Process Proof

In the above screenshot, similar to our first, we see a console logged process for proof of the cryptographic exchange. On login, we see a new ECDH key pair generated for both alice and bob (noted as "Generated new ECDH Pair"), we see an output of the shared secret derivation process where each party gets the others public key to encrypt / decrypt the AES key, using the AES key along with generating a nonce in the sender's terminal (bob), and seeing the nonce and ciphertext in the server terminal on the left. Lastly, the second image shows that even when the same message is sent twice, a different ciphertext is generated.

5. Integration Proof



The above four screenshots show a successful integration proof in the messengers current state. The

first screenshot shows both the GitHubOAuth and Challenge-response login working and generating a new ECDH pair along with a message exchange, the second screenshot shows multiple users sending messages to each other at once, the third screenshot is again our encrypted message log database, and the fourth screenshot shows user management features working alongside E2EE which in that case was using the list users function with an admin account while a live exchange was happening.

6. Security Properties Verification

Finally, the above two screenshots demonstrate our security property verification with a demonstration of non-refreshed public keys on one users terminal while the other had generated a new pair on login, meaning when a message was sent we see the inegrity error printed to console. The second screenshot shows our session expiration and connection / key cleannup helping support forward secrecy for users by preventing a malicious actor from recycling old keys from previous sessions, setting a timeout on inactivity of 30 minutes.

---

5. Threat Model

Use the following security properties and threat actors in your threat modeling. You can add extra if needed.

Threat Actors: 1. **Passive Network Attacker**: Can intercept but not modify or read traffic 2.

Active Network Attacker: Can intercept but not read or modify traffic 3. **Malicious Server Operator**: Has access to server and database, but none of messages exchanged between users or their credentials are exposed internally. 4. **Compromised Client**: Attacker has access to a single user's device, but cannot gain access to other user accounts or read any confidential information along with session expiry and forced re-authentication after 30 minutes of inactivity.

Security Properties Achieved: - [X] Confidentiality - [X] Integrity
- [X] Authentication - [X] Authorization - [X] Non-repudiation - [ ] Perfect Forward Secrecy (some forward secrecy is attained) - [X] Privacy

Before vs. After Analysis: - **Authentication**: [If applicable otherwise remove][Improved by verifying messages with non-repudiation trhough ECDH exchange] - **Authorization**: [If applicable otherwise remove][Only authorized users i.e. intended recipients can read and decrypt messages]
- **Data Protection**: [If applicable otherwise remove][Message integrity and privacy obtained between users] - **Communication Security**: [If applicable otherwise remove][E2EE with ECDH and AES provide security on message interception and tampering]

---

6. Performance Analysis

6.1 System Performance with E2EE Enabled

With E2EE enabled, the performance of our application noticeably saw a decline. The messages between users was slower, along with the application being a lot more unstable with connections likely due to flawed / buggy code implementation as this is my first attempt / iteration of implementing these properties but along with the several requests going back and forth between client terminals and the server. Not to mention, the application performance overall has become much less convenient to a user. Forced re-authentication on 30- minutes of inactivity, limits to the number of messages they can send along with commands, and the authentication at any sensitive step, makes it much more difficult to use the full extent of the applications elements.

---

7. Conclusion

7.1 Summary of Achievements

Over the course of this assignment, and the previous two prior, I have significantly increased the security of the python messaging application we were initially provided. From securing passwords at rest, MAC implementations, TOTP, OAuth, and challenge-response based authentication methods, role-based access for sensitive actions, logging, and finally E2EE and forward-secrecy, our application is significantly more secure than its original state and I have learned a lot about the difficulties and large amount of time required to implement security solutions into many modern applications.

7.2 Security and Privacy Posture Assessment

The final implementation is fairly secure, while some acknowledged security vulnerabilities do exist in it still for the sake of our demo environment and assignment ease, the many security methods introduced have helped reduce the many vulnerabilities that existed in assignment #1.

Remaining Vulnerabilities: - Vulnerability 1: [Minimal input validation from users inputted messages or login methods, in theory a user can insert malicious scripts or codes to attempt to bypass or force certain sensitive action] - Vulnerability 2: [Open client-server communication over TCP sockets]

Suggest an Attack: A malicious actor could make a legitimate account and attempt a DoS or bypass through the applications lack of input validation to elevate privilege or gain access to sensitive information.

7.3 Future Improvements

1. **Improvement 1**: [Proper implementation of shared key, FERNET key, and client ID / client secret for in-app credential storage]
2. **Improvement 2**: [Fix the timeout loop caused in OAuth and Challenge-response based re-authentication]

---

8. References

Github Co-Pilot to assist in troubleshooting and implementation for key exchange, and clear session debugging to identify issues in session management and proper client messaging exchange. Provided me with walkthroughs on correct implementation of large portion of E2EE along with the best additions to log to console.

Submission Checklist

Before submitting, ensure you have:

- ☐ **Complete Report**: All sections filled out with sufficient detail
- ☐ **Evidence**: Screenshots, logs, and demonstrations included
- ☐ **Code**: Well-named(based on task and whether it is an attack or a fix) and well-commented and organized in your GitHub repository deliverable directory of the corresponding assignment
- ☐ **Tests**: Security and functionality tests implemented after fix
- ☐ **GitHub Link**: Repository link included in report and Brightspace submission
- ☐ **Academic Integrity**: All sources properly cited, work is your own

---

Submission Instructions: 1. Save this report as PDF: `[StudentID]_Assignment[X]_Report.pdf` 2. Submit PDF to Brightspace 3. Include your GitHub repository fork link in the Brightspace submission comments 4. Ensure your repository is private until after course completion otherwise you'll get zero grade

Final Notes: - Use **GenAI** for help but do not let **GenAI** to do all the work and you should understand everything yourself - If you used any **GenAI** help make sure you cite the contribution of **GenAI** properly - Be honest about limitations and challenges - Focus on demonstrating understanding, not just working code - Quality over quantity - depth of analysis is more important than length - Proofread for clarity and technical accuracy

Good luck!