Course: ECE 572; Summer 2025

Instructor: Dr. Ardeshir Shojaeinasab

Student Name: Yujian Li
Student ID: V01046555
Assignment: [Assignment 3]
Date: Aug 4th
GitHub Repository: https://github.com/YujianLiG208/ECE572

Executive Summary

implement end-to-end encryption (E2EE) with session management.

Table of Contents

---

1. Introduction

1.1 Objective

Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol

Hybrid encryption using ECDH + AES-GCM

Session management and expiration mechanisms

End-to-end encryption security properties

Key management and secure session cleanup

1.2 Scope

N/A

1.3 Environment Setup

• Operating System: WIN 10, with WSL2 Ubuntu 24.04

• Python Version: 3.13.0

• Key Libraries Used: (only new libraries) threading, cryptography, base64

• Development Tools: Visual Studio Code with Copilot, ChatGPT, Google Authenticator. Codes are originally generated with ChatGPT, optimized and modified by myself with help of Copilot.

2. Task Implementation

2.1 Task X: ECDH Key Exchange

2.1.1 Objective

Apply ECDH key exchange protocol

2.1.2 Implementation Details

Key Components: -

A. [Libraries]

cryptography.hazmat.primitives.asymmetric.ec - For elliptic curve operations

cryptography.hazmat.primitives.kdf.hkdf - For key derivation (HKDF-SHA256)

cryptography.hazmat.primitives.ciphers.aead - For AES-GCM encryption

cryptography.hazmat.primitives.serialization - For key serialization

B. [Core Data Structures]

self.user_pubkeys = {} - Server storage for user public keys (username → PEM-encoded public key)

self.session_keys = {} - Server storage for derived session keys (username → {peer_username → derived_key})

self.ecdh_private_key - Client's private ECDH key

self.ecdh_public_key_pem - Client's public key in PEM format

C. [Key Exchange Protocol Components]

Key Generation - SECP256R1 elliptic curve

Public Key Distribution - Server-mediated key exchange

Shared Secret Derivation - ECDH + HKDF-SHA256

Message Encryption - AES-256-GCM with derived keys

Code Snippet (Key Implementation):

[ECDH KEY GENERATION]

```
def generate_ecdh_keypair(self):
    """Generate ECDH key pair and store PEM public key."""
    self.ecdh_private_key = ec.generate_private_key(ec.SECP256R1())
    pubkey = self.ecdh_private_key.public_key()
    self.ecdh_public_key_pem = pubkey.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ).decode('utf-8')

    print(f"[ECDH] Generated new ECDH key pair for {self.username}")
```

[Public Key Publishing to Server]

```
def publish_pubkey(self):
    """Send public key to server after login."""
    command = {
        'command': 'PUBLISH_PUBKEY',
        'username': self.username,
        'pubkey': self.ecdh_public_key_pem
    }
    response = self.send_command(command)
```

[Server-Side Public Key Storage]

```
elif command == 'PUBLISH_PUBKEY':
    username = message.get('username')
    pubkey_pem = message.get('pubkey')
    self.store_user_pubkey(username, pubkey_pem)
    response = {'status': 'success', 'message': 'Public key stored'}

def store_user_pubkey(self, username, pubkey_pem):
    """Store a user's public key (PEM format)."""
    self.user_pubkeys[username] = pubkey_pem
```

[Peer Public Key Retrieval]

```
def get_peer_pubkey(self, peer_username):
    """Get another user's public key from the server."""
    command = {
        'command': 'GET_PUBKEY',
        'target_user': peer_username
    }
    response = self.send_command(command)

    if response.get('status') == 'success':
        try:
            pubkey =
serialization.load_pem_public_key(response['pubkey'].encode('utf-8'))
            return pubkey
        except Exception as e:
            print(f"Error loading public key: {e}")
            return None
```

[Shared Secret Derivation with HKDF]

```
def derive_shared_key(self, peer_pubkey):
    """Derive shared key using ECDH and return 32-byte key for AES-256."""
    shared_key = self.ecdh_private_key.exchange(ec.ECDH(), peer_pubkey)
    # Use HKDF to derive a proper 32-byte key
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,  # 32 bytes for AES-256
        salt=None,
        info=b'SecureText ECDH',
    ).derive(shared_key)

    return derived_key
```

[Message Encryption with Derived Key]

```
def encrypt_message(self, plaintext, shared_key):
    """Encrypt a message using AES-256-GCM."""
    aesgcm = AESGCM(shared_key)
    nonce = os.urandom(12)  # 96-bit nonce for GCM
    ciphertext = aesgcm.encrypt(nonce, plaintext.encode('utf-8'), None)
    # Send nonce + ciphertext, both base64-encoded for JSON safety
    encrypted = base64.b64encode(nonce + ciphertext).decode('utf-8')
```

```
    return encrypted
```

[Message Decryption]

```
def decrypt_message(self, b64_ciphertext, shared_key):
    """Decrypt a message using AES-256-GCM."""
    aesgcm = AESGCM(shared_key)
    data = base64.b64decode(b64_ciphertext)
    nonce = data[:12]
    ciphertext = data[12:]
    plaintext = aesgcm.decrypt(nonce, ciphertext, None)
    return plaintext.decode('utf-8')
```

2.1.3 Challenges and Solutions

2.1.3.1. [Key Management and Storage Challenges]

Challenge: Public Key Distribution

Problem: Secure distribution of ECDH public keys between clients through the server

Evidence: Evidence: Server storage via self.user_pubkeys = {} and commands PUBLISH_PUBKEY/GET_PUBKEY

Solution:

```
def store_user_pubkey(self, username, pubkey_pem):
    """Store a user's public key (PEM format)."""
    self.user_pubkeys[username] = pubkey_pem

    # Enhanced logging for ECDH key generation
    self.log_inspection_event(
        "ECDH_PUBKEY_STORED",
        username,
        f"ECDH public key generated and stored for {username}",
        {
            "key_algorithm": "SECP256R1",
            "key_format": "PEM",
            "key_length": len(pubkey_pem)
        }
    )
```

2.1.3.2. [Error Handling and Resilience]

Challenge: Invalid Response Handling

Problem: Network communication failures and malformed responses

Evidence: Multiple error checks in client code

Solution:

```
def get_peer_pubkey(self, peer_username):
    # Add error handling for missing 'status' key
    if not isinstance(response, dict):
        print(f"Error: Invalid response from server: {response}")
        return None

    if response.get('status') == 'success':
        try:
            pubkey =
serialization.load_pem_public_key(response['pubkey'].encode('utf-8'))
            return pubkey
        except Exception as e:
            print(f"Error loading public key: {e}")
            return None
    else:
        print(f"Error: {response.get('message', 'Unknown error
occurred')}")
        return None
```

2.1.3.3 [Cryptographic Implementation Challenges]

Challenge: Key Derivation from ECDH Shared Secret

Problem: Raw ECDH output isn't suitable for direct use as encryption key

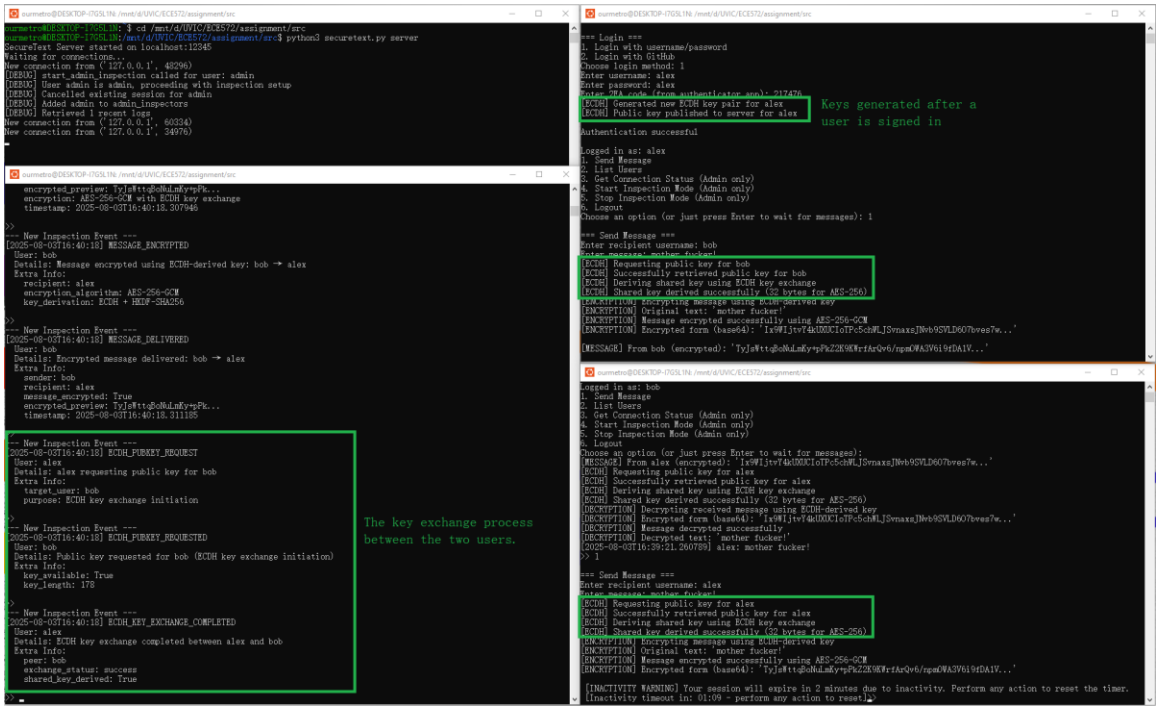
Solution:

```
def derive_shared_key(self, peer_pubkey):
    """Derive shared key using ECDH and return 32-byte key for AES-256."""
    shared_key = self.ecdh_private_key.exchange(ec.ECDH(), peer_pubkey)
    # Use HKDF to derive a proper 32-byte key
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,  # 32 bytes for AES-256
        salt=None,
        info=b'SecureText ECDH',
```

```
    ).derive(shared_key)
    return derived_key
```

2.1.4 Testing and Validation

note: this test is done after the admin inspection mode in task Z is finished.



(You might need to zoom up to see the screenshot clearly)

Keys are generated after a user is signed in. Each time they send messages, a key exchange is required to ensure confidentiality. ECDH keys exchange process and the encrypted key is clearly shown on the lower left hand side in the admin's inspection mode.

For demonstration purpose, the keys is visible to the admin. However, in a business environment, it should not.

2.2 Task Y: AES-256-GCM Encryption

2.2.1 Objective

Use AES-256-GCM for authenticated encryption, generate unique nonces for each message.

2.2.2 Implementation Details

A. [Libraries]

from cryptography.hazmat.primitives.ciphers.aead import AESGCM

import base64

B. [Core Data Structures]

Shared Keys: Derived from ECDH key exchange (32 bytes for AES-256)

Nonces: 96-bit (12 bytes) randomly generated for each message

Ciphertext Format: nonce + encrypted_data + auth_tag (all combined)

C. [Message Format Structure Example]

Original Message: "Hello World"

↓

UTF-8 Encoding: byte array

↓

AES-256-GCM Encryption: nonce(12 bytes) + ciphertext + auth_tag

↓

Base64 Encoding: "abc123def456..." (for transport)

Code Snippet (Key Implementation):

Encryption and Decryption with AES-256-GCM

```
def encrypt_message(self, plaintext, shared_key):
    """Encrypt a message using AES-256-GCM."""
    print(f"[ENCRYPTION] Encrypting message using ECDH-derived key")
    print(f"[ENCRYPTION] Original text: '{plaintext}'")

    aesgcm = AESGCM(shared_key)
    nonce = os.urandom(12)  # 96-bit nonce for GCM
    ciphertext = aesgcm.encrypt(nonce, plaintext.encode('utf-8'), None)
    # Send nonce + ciphertext, both base64-encoded for JSON safety
    encrypted = base64.b64encode(nonce + ciphertext).decode('utf-8')

    print(f"[ENCRYPTION] Message encrypted successfully using AES-256-GCM")
    print(f"[ENCRYPTION] Encrypted form (base64): '{encrypted[:50]}...'")
    return encrypted
```

(Decryption is on the next page)

```
def decrypt_message(self, b64_ciphertext, shared_key):
    """Decrypt a message using AES-256-GCM."""
    print(f"[DECRYPTION] Decrypting received message using ECDH-derived
key")
    print(f"[DECRYPTION] Encrypted form (base64):
'{b64_ciphertext[:50]}...'")

    aesgcm = AESGCM(shared_key)
    data = base64.b64decode(b64_ciphertext)
    nonce = data[:12]
    ciphertext = data[12:]
    plaintext = aesgcm.decrypt(nonce, ciphertext, None)

    decoded_text = plaintext.decode('utf-8')
    print(f"[DECRYPTION] Message decrypted successfully")
    print(f"[DECRYPTION] Decrypted text: '{decoded_text}'")
    return decoded_text
```

Key Derivation for AES-256 (32 byte)

```
def derive_shared_key(self, peer_pubkey):
    """Derive shared key using ECDH and return 32-byte key for AES-256."""
    shared_key = self.ecdh_private_key.exchange(ec.ECDH(), peer_pubkey)
    # Use HKDF to derive a proper 32-byte key
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,  # 32 bytes for AES-256
        salt=None,
        info=b'SecureText ECDH',
    ).derive(shared_key)

    print(f"[ECDH] Shared key derived successfully (32 bytes for AES-256)")
    return derived_key
```

Send Message with Encryption

```
def send_message(self):
    """Send a message to another user (ECDH key exchange for
encryption)"""
    # ... input validation ...
```

```
    try:
        # --- ECDH Key Exchange ---
        peer_pubkey = self.get_peer_pubkey(recipient)
        if not peer_pubkey:
            print("Could not get recipient's public key.")
            return

        shared_key = self.derive_shared_key(peer_pubkey)
        encrypted_content = self.encrypt_message(content, shared_key)
        # -------------------------

        command = {
            'command': 'SEND_MESSAGE',
            'recipient': recipient,
            'content': encrypted_content  # Encrypted!
        }

        response = self.send_command(command)
```

Receive Message with Decryption

```
# In listen_for_messages method
if message.get('type') == 'MESSAGE':
    sender = message['from']
    encrypted_content = message['content']

    print(f"\n[MESSAGE] From {sender} (encrypted):
'{encrypted_content[:50]}...'")

    # Get peer's public key and derive shared key
    peer_pubkey = self.get_peer_pubkey(sender)
    if peer_pubkey:
        shared_key = self.derive_shared_key(peer_pubkey)
        plaintext = self.decrypt_message(encrypted_content, shared_key)
        print(f"[MESSAGE] {sender}: {plaintext}")
```

2.2.3 Challenges and Solutions

Nonce Management and Uniqueness

AES-GCM requires that each encryption operation uses a unique nonce (number used once). Reusing nonces with the same key completely breaks the security of GCM mode, potentially allowing attackers to:

Recover plaintext messages

Forge authentication tags

Break confidentiality and authenticity guarantees

Solution:

```
def encrypt_message(self, plaintext, shared_key):
    """Encrypt a message using AES-256-GCM."""
    aesgcm = AESGCM(shared_key)
    nonce = os.urandom(12)  # 96-bit nonce for GCM - UNIQUE PER MESSAGE
    ciphertext = aesgcm.encrypt(nonce, plaintext.encode('utf-8'), None)
    # Send nonce + ciphertext, both base64-encoded for JSON safety
    encrypted = base64.b64encode(nonce + ciphertext).decode('utf-8')
    return encrypted
```

Solution Elements:

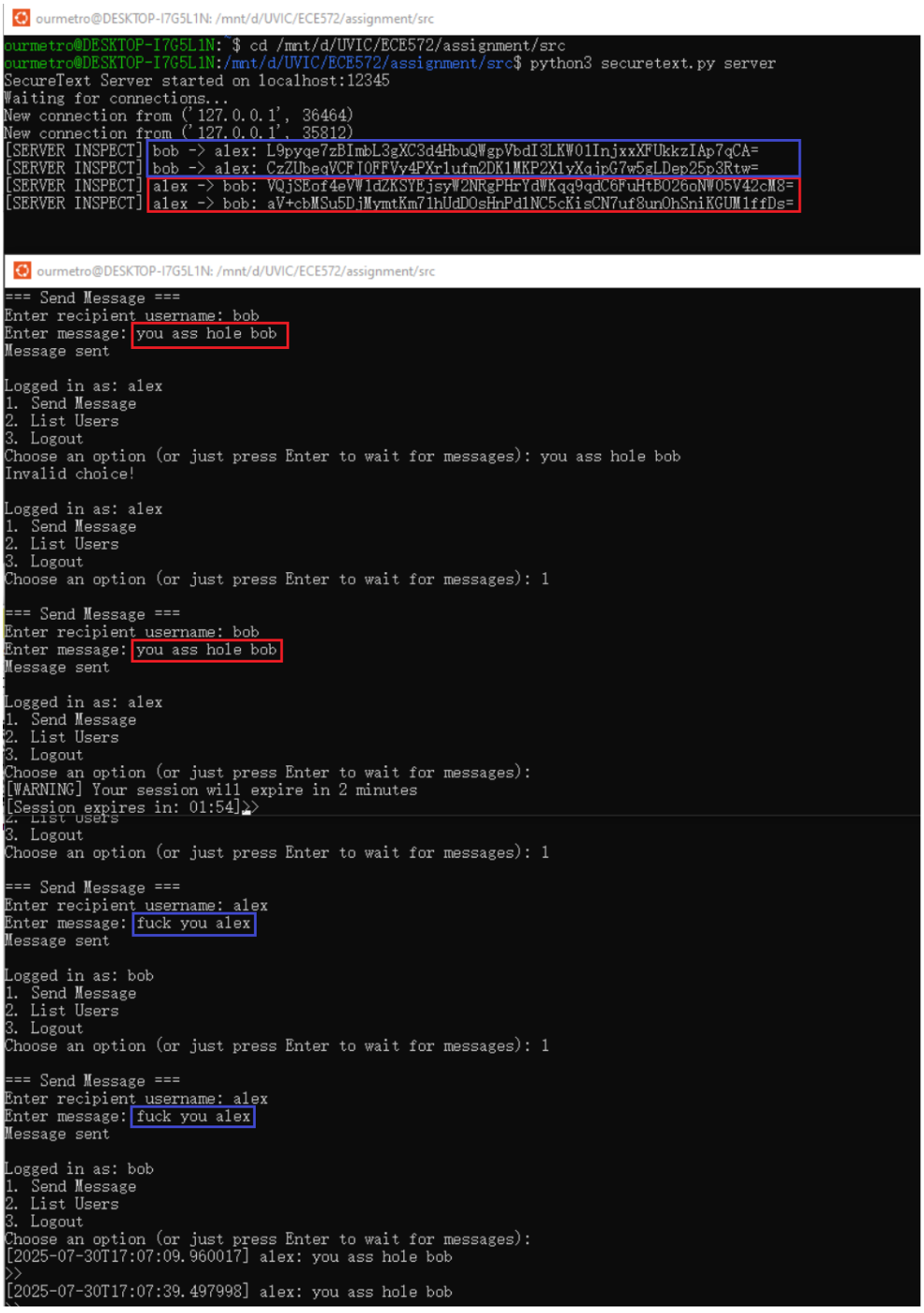
Cryptographically Secure Random Generation: os.urandom(12) ensures each nonce is unpredictable and unique

Proper Size: 96-bit (12 bytes) nonces are optimal for GCM mode

Per-Message Generation: Fresh nonce generated for every single encryption operation

Nonce Transmission: Nonce is prepended to ciphertext for decryption

2.2.4 Testing and Validation



Let's test the implementation.

Alex sent two "you ass hole bob" to bob; Bob sent two "fuck you alex" to alex. Same message produces different ciphertext each time.

On the server side, we can see that the messages are encrypted and unreadable, which means that server only routes encrypted data.

2.3 Task Z: Session Management & Integration with existing functions

2.3.1 Objective

Implement session time out and encryption tool clean up after session timeout, ensure administrator have a real time view on what is happening (admin does not have time out in inspection mode).

NOTE: The session timeout requirement on Github is 30 min, in order to reduce waiting time during development/testing/demonstration, I set it to 5 mins.

2.3.2 Implementation Details

A. Role-Based Access Control (RBAC)

Admin Flag: Users have an admin boolean flag in their profile

Privilege Verification: Admin functions check user roles before execution

Privilege Escalation: Admins bypass normal session timeouts during inspection

B. Session Management Security

Inactivity Timeout: Configurable session expiration (5 minutes default)

Warning System: Users get 2-minute warning before expiration

Activity-Based Reset: Any user action resets the session timer

Comprehensive Cleanup: Sessions clear all cryptographic material on expiration

C. Real-time Monitoring

Inspection Events: All security events logged for admin review

Live Notifications: Admins receive real-time event notifications

Threaded Delivery: Non-blocking notification system

Event Buffering: Maintains last 100 inspection events

D. Integration Points

Authentication Integration: Session creation happens after successful login

Command Integration: Session reset occurs on every client command

Cryptographic Integration: Session cleanup removes ECDH keys and derived secrets

Audit Integration: Session events logged for security audit trail

[Session Timeout Management]

```
SESSION_TIMEOUT_MINUTES = 5  # Configurable for testing
WARNING_BEFORE_TIMEOUT_MINUTES = 2  # User warning system

# Server Data Structures
self.sessions = {}  # username -> {timestamp, timer}
self.admin_inspectors = {}  # username -> connection (admins in inspection
mode)
self.inspection_logs = []  # Store recent inspection events
self.max_inspection_logs = 100  # Keep last 100 events

def expire_session(self, username):
    """Handle session expiration with comprehensive cleanup"""
    # Log session expiration BEFORE cleanup for admin inspection
    self.log_inspection_event(
        "SESSION_TIMEOUT",
        username,
        f"SESSION EXPIRED: User {username} session expired after
{SESSION_TIMEOUT_MINUTES} minutes"
    )
```

[Session Creation and Timer Management]

```
def create_session(self, username):
    """Create a new session for user with timeout"""
    if username in self.sessions:
        self.clear_session(username)

    # Create warning and timeout timers
    warning_timer = Timer(
        (SESSION_TIMEOUT_MINUTES - WARNING_BEFORE_TIMEOUT_MINUTES) * 60,
        self.send_timeout_warning,
        args=[username]
    )
    timeout_timer = Timer(
        SESSION_TIMEOUT_MINUTES * 60,
        self.expire_session,
        args=[username]
    )

    self.sessions[username] = {
        'timestamp': time.time(),
        'warning_timer': warning_timer,
```

```
        'timeout_timer': timeout_timer,
        'last_activity': time.time()  # Track last activity
    }

    # Start the timers
    warning_timer.start()
    timeout_timer.start()
```

[Session Activity Reset on User Actions]

```
def reset_session_timer(self, username):
    """Reset the session timeout for a user upon activity."""
    if username in self.sessions:
        # Cancel existing timers
        self.sessions[username]['warning_timer'].cancel()
        self.sessions[username]['timeout_timer'].cancel()

        # Create new timers
        warning_timer = Timer(
            (SESSION_TIMEOUT_MINUTES - WARNING_BEFORE_TIMEOUT_MINUTES) *
60,
            self.send_timeout_warning,
            args=[username]
        )
        timeout_timer = Timer(
            SESSION_TIMEOUT_MINUTES * 60,
            self.expire_session,
            args=[username]
        )

        # Update session with new timers and activity timestamp
        self.sessions[username]['warning_timer'] = warning_timer
        self.sessions[username]['timeout_timer'] = timeout_timer
        self.sessions[username]['last_activity'] = time.time()

        # Start the new timers
        warning_timer.start()
        timeout_timer.start()
```

[Session Expiration with Comprehensive Cleanup]

```
def expire_session(self, username):
    """Handle session expiration with comprehensive cleanup"""
    print(f"[SESSION] Expiring session for user: {username} due to
inactivity")

    # Log session expiration BEFORE cleanup for admin inspection
    self.log_inspection_event(
        "SESSION_TIMEOUT",
        username,
        f"SESSION EXPIRED: User {username} session expired after
{SESSION_TIMEOUT_MINUTES} minutes of inactivity",
        {
            "timeout_duration_minutes": SESSION_TIMEOUT_MINUTES,
            "expiration_reason": "inactivity_timeout",
            "cleanup_pending": True,
            "user_notified": username in self.active_connections,
            "last_activity": self.sessions.get(username,
{}).get('last_activity', 'unknown')
        }
    )

    if username in self.active_connections:
        expiration_msg = {
            'type': 'SESSION_EXPIRED',
            'message': f'Your session has expired due to
{SESSION_TIMEOUT_MINUTES} minutes of inactivity. Please login again.',
            'force_cleanup': True
        }
        try:
            self.active_connections[username].send(
                json.dumps(expiration_msg).encode('utf-8')
            )
            print(f"[SESSION] Sent inactivity-based session expiration
notice to {username}")
        except:
            print(f"[SESSION] Could not notify {username} of session
expiration")

    # Clean up all session data and cryptographic material
    if username in self.active_connections:
        try:
            self.active_connections[username].close()
        except:
```

```
            pass
        del self.active_connections[username]
        print(f"[SESSION] Closed connection for {username}")

    # Clear all session data including keys and messages
    self.clear_session(username)

    # Log the session expiration for security audit
    self.log_action("session", username, f"Session expired after
{SESSION_TIMEOUT_MINUTES} minutes of inactivity - all keys and data
cleared", "expired")
```

[Admin inspection mode & Event logging]

```
def start_admin_inspection(self, admin_username, conn):
    """Start admin inspection mode - no session timeout"""
    print(f"[DEBUG] start_admin_inspection called for user:
{admin_username}")

    # Role-based access control
    if not self.users.get(admin_username, {}).get('admin', False):
        print(f"[DEBUG] User {admin_username} is not admin")
        return False, "Admin privileges required", None

    print(f"[DEBUG] User {admin_username} is admin, proceeding with
inspection setup")

    # Cancel any existing session timeout for this admin - PRIVILEGE
ESCALATION
    if admin_username in self.sessions:
        self.sessions[admin_username]['warning_timer'].cancel()
        self.sessions[admin_username]['timeout_timer'].cancel()
        del self.sessions[admin_username]
        print(f"[DEBUG] Cancelled existing session for {admin_username}")

    # Add to admin inspectors list
    self.admin_inspectors[admin_username] = conn
    print(f"[DEBUG] Added {admin_username} to admin_inspectors")

    # Send recent inspection logs to admin
    try:
        recent_logs = self.get_recent_inspection_logs()
```

```
        print(f"[DEBUG] Retrieved {len(recent_logs)} recent logs")
        return True, "Inspection mode activated", recent_logs
    except Exception as e:
        print(f"[DEBUG] Error getting recent logs: {e}")
        return True, "Inspection mode activated", []
def log_inspection_event(self, event_type, username, details,
extra_data=None):
    """Log inspection events for admin monitoring"""
    event = {
        'timestamp': datetime.now().isoformat(),
        'event_type': event_type,
        'username': username,
        'details': details,
        'extra_data': extra_data
    }

    self.inspection_logs.append(event)

    # Keep only recent logs
    if len(self.inspection_logs) > self.max_inspection_logs:
        self.inspection_logs = self.inspection_logs[-
self.max_inspection_logs:]

    # Notify all active admin inspectors
    self.notify_admin_inspectors(event)
```

[Session Activity Integration in Client Handler]

```
def handle_client(self, conn, addr):
    """Handle individual client connection"""
    try:
        while True:
            data = conn.recv(8192).decode('utf-8')
            # ... data processing ...

            # Reset session timer IMMEDIATELY when receiving data from
logged-in users
            # This happens BEFORE command processing to ensure timer reset
happens first
            if current_user and current_user in self.sessions:
                self.reset_session_timer(current_user)
```

```
            # ... command processing ...
    finally:
        # Clean up connection and session
        if current_user:
            if current_user in self.active_connections:
                del self.active_connections[current_user]
            # Ensure session is cleared on disconnect
            if current_user in self.sessions:
                self.clear_session(current_user)
```

2.3.3 Challenges and Solutions

2.3.3.1 [Session Management and Key Cleanup]

Challenge: Forward Secrecy and Key Lifecycle

Problem: Ensuring ephemeral keys are properly destroyed on session end

Solution: Comprehensive cleanup in clear_session()

```
def clear_session(self, username):
    """Safely clear session data and cryptographic material"""
    # Clear user's public key from server storage
    if username in self.user_pubkeys:
        del self.user_pubkeys[username]
        cleared_items.append("public_key")

    # Clear all session keys involving this user
    if username in self.session_keys:
        key_count = len(self.session_keys[username])
        del self.session_keys[username]
        cleared_items.append(f"session_keys({key_count})")
```

2.3.3.2 [Admin Session Timeout Conflicts with Inspection Mode]

The Problem:

One of the most significant challenges in implementing Session Management with role-based Admin Inspection was preventing admin users from being automatically logged out due to session timeouts while they are in inspection mode.

The system has automatic session expiration after 5 minutes of inactivity, which works well for regular users. However, admin users performing inspection duties might need to monitor system events for extended periods without actively sending

commands, which would normally trigger a session timeout and force them to re-authenticate.

The Conflict:

Regular Session Management: Users are automatically logged out after SESSION_TIMEOUT_MINUTES (5 minutes) of inactivity

Admin Inspection Requirements: Admins need to monitor real-time events without being forced to re-authenticate

Security vs. Usability: Can't disable timeouts entirely (security risk) but can't interrupt critical monitoring (usability issue)

The Solution:

The solution implemented uses selective session timeout cancellation for admin users in inspection mode:

```
def start_admin_inspection(self, admin_username, conn):
    """Start admin inspection mode - no session timeout"""
    print(f"[DEBUG] start_admin_inspection called for user:
{admin_username}")

    if not self.users.get(admin_username, {}).get('admin', False):
        print(f"[DEBUG] User {admin_username} is not admin")
        return False, "Admin privileges required", None

    print(f"[DEBUG] User {admin_username} is admin, proceeding with
inspection setup")

    # Cancel any existing session timeout for this admin
    if admin_username in self.sessions:
        self.sessions[admin_username]['warning_timer'].cancel()
        self.sessions[admin_username]['timeout_timer'].cancel()
        del self.sessions[admin_username]
        print(f"[DEBUG] Cancelled existing session for {admin_username}")

    self.admin_inspectors[admin_username] = conn
    # No new session timers created - admin stays logged in indefinitely
```

And when inspection mode ends, normal session management is restored:

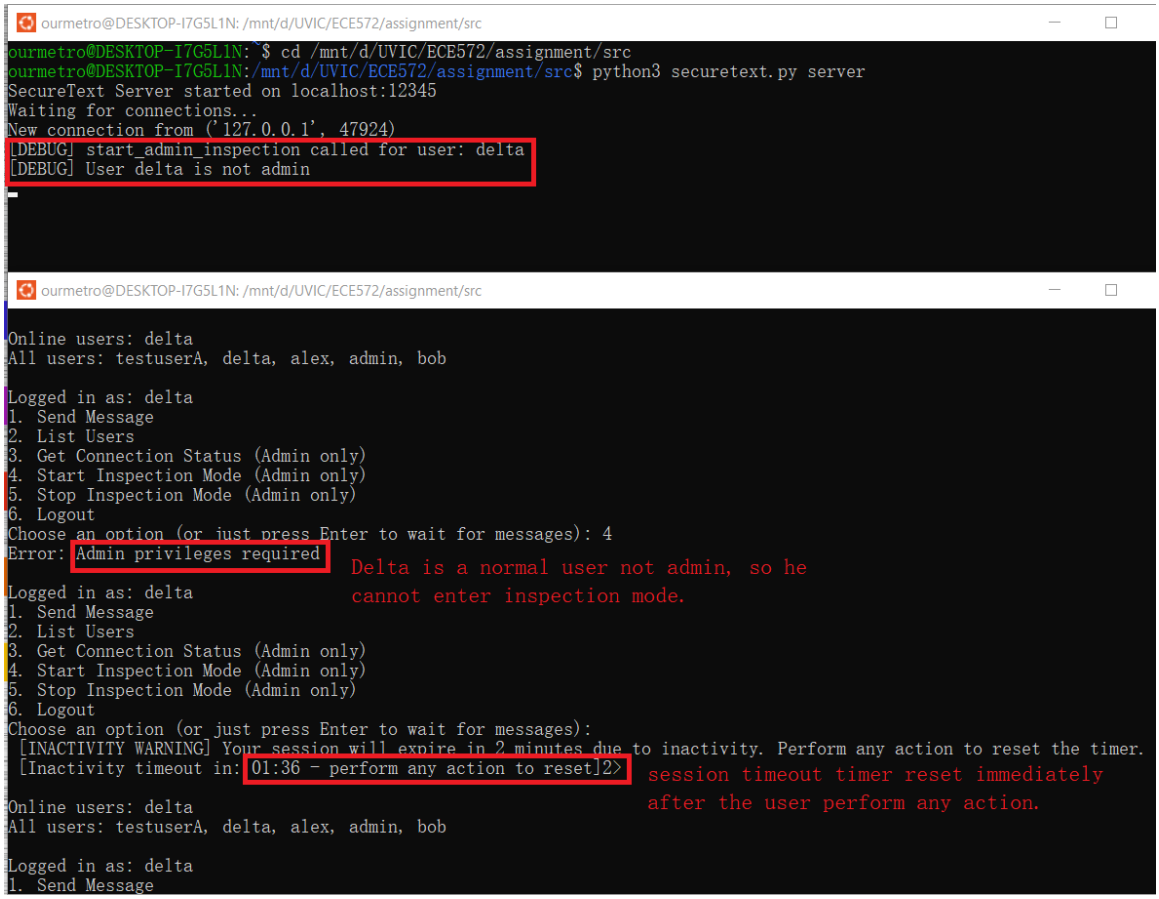
```
def stop_admin_inspection(self, admin_username):
    """Stop admin inspection mode and restore normal session"""
    if admin_username in self.admin_inspectors:
```

```
        del self.admin_inspectors[admin_username]
        self.log_inspection_event("ADMIN_INSPECTION_STOP", admin_username,
"Admin stopped inspection mode")
        # Restart normal session
        self.create_session(admin_username)  # Restore timeout timers
        return True, "Inspection mode deactivated"
    return False, "Not in inspection mode"
```

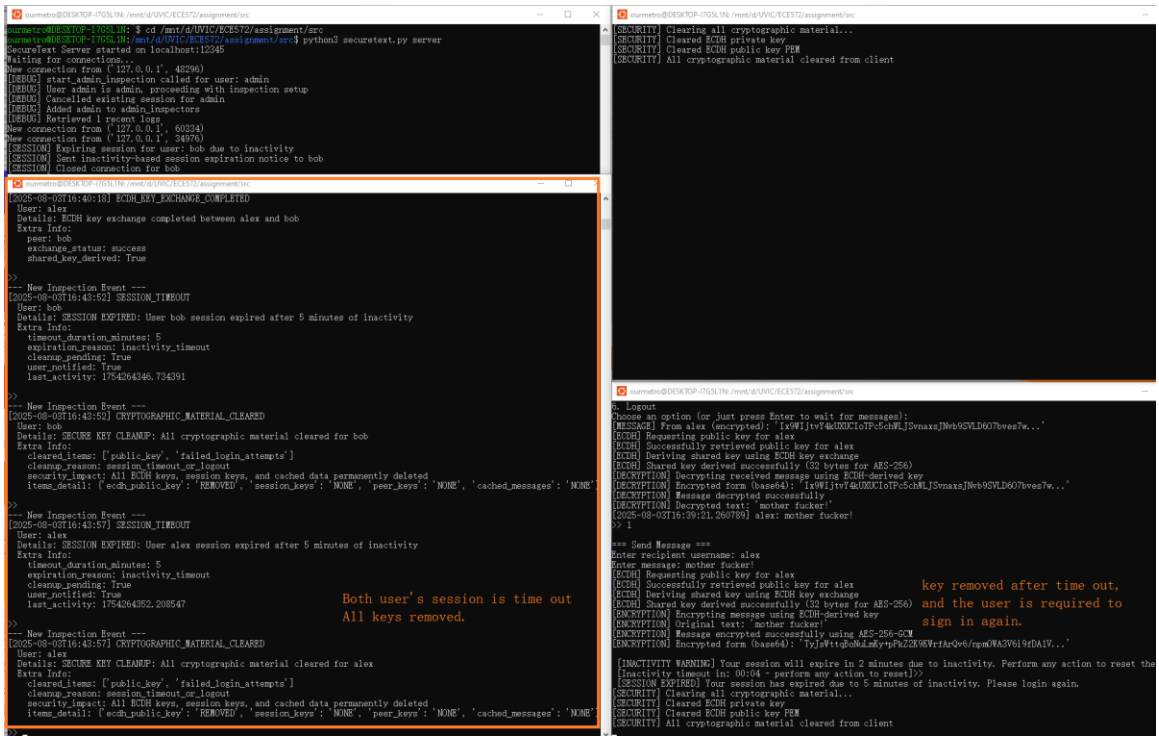
Timeout timers are cancelled when entering inspection mode, admin inspectors are tracked separately from regular sessions.

This solution ensures that admin users can perform their monitoring duties without interruption while maintaining the security benefits of automatic session timeouts for regular users and admins not in inspection mode.

2.3.4 Testing and Validation



Let's test the implementation. As shown in the screenshot, a normal user has no access to perform admin-only actions. I signed in as delta, do nothing for 3 minutes, after the timer pops up, perform any action would stop it from keep counting, and the session time out is reset.



(You might need to zoom up to see the screenshot clearly)

On the lower left-hand side, is the administrator in inspection mode, he won't have timeout problem in this mode unless he quit by himself.

We can see that after the two user's session get time out because of no action, the keys are cleaned. To continue sending messages, use alex and bob needs to sign in again, they will then get a new key after their new session starts.

---

3. Security Analysis

3.1 Security Improvements

Confidentiality: Server cannot read message content

Integrity: Message tampering is impossible

Authentication: Verify message sender identity

Forward Secrecy: Session expiration limits compromise impact

Session Security: Proper timeout and cleanup behavior

3.2 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 & decrypt traffic

2. **Active Network Attacker**: Can not intercept and modify traffic

3. **Malicious Server Operator**: Has no access to server and database

4. **Compromised Client**: Attacker has no access to user's device. Even if the user's device is compromised, they won't be able to sign in.

Security Properties Achieved:

- [√] Confidentiality: AES-256-GCM encryption with ECDH key exchange.

- [√] Integrity: AES-GCM authenticated encryption, HMAC-SHA256 challenge-response.

-[√] Authentication: Already implemented in the previous assignment, but fixed to make it actually working in this one.

- [ ] Authorization: Role-based access control (admin vs regular users)

- [ ] Non-repudiation: N/A

- [√] Perfect Forward Secrecy: Ephemeral ECDH keys destroyed on session timeout/logout

- [√] Privacy: End-to-end encryption, session management, cryptographic material cleanup

---

4. Attack Demonstrations

(Not applicable for this assignment requirement)

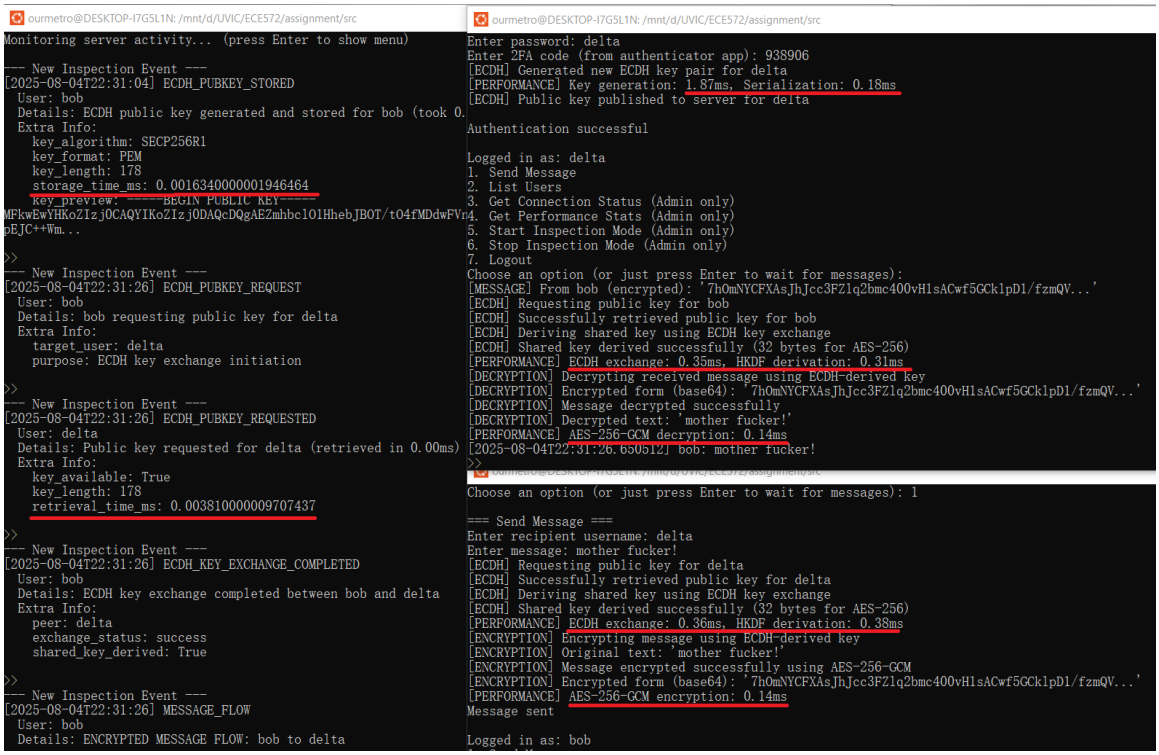
---

5. Performance Evaluation

No noticeable performance issues in development and test environments (my laptop), after E2EE implementation, there is no noticeable major downgrade.

However, to ensure the rigor of the implementation, I modified a special version of securetext (securetext_pe.py), use the psutil packet to monitor and measuring the performance of my implemented solutions.

The screenshot is on the next page, you might also want to zoom up to have a clear look on it.



As the result shows, ECDH key generation, storage and exchange, as well as message encryption and decryption, were all incredibly fast, with virtually no impact on performance. Therefore, I can confidently say that all the solutions implemented above were successful.

However, when I tried to open multiple WSL Ubuntu terminals for multi-user testing and real-time monitoring, after I opened the fifth terminal, the system would always get stuck. Sometimes I had to hard shut down and restart to alleviate it. The cause of this problem has not yet been found.

---

6. Lessons Learned

6.1 Technical Insights

Proper key lifecycle management: generation, distribution, storage, rotation, destruction is as important as the cryptographic algorithms themselves.

Sessions must be properly managed with timeouts, activity tracking, and secure cleanup to prevent session hijacking and resource leaks.

When errors occur, the system should default to a secure state rather than an insecure one

6.2 Security Principles

Applied Principles:

Defense in Depth: implemented multiple layers of security including password hashing with Argon2, TOTP 2FA, ECDH key exchange, AES-256-GCM encryption, session timeouts, challenge-response authentication, and comprehensive audit logging to ensure that if one security measure fails, others remain to protect the system.

Least Privilege: implemented role-based access control where admin functions (like inspection mode, connection status, and password resets for other users) are restricted to users with the admin flag set to True, ensuring regular users cannot access administrative capabilities.

Fail Secure: modified the system to default to secure states when errors occur - invalid authentication attempts result in login denial, missing public keys prevent message encryption, session timeouts automatically clear all cryptographic material, and communication errors return error messages rather than allowing unauthorized access.

Economy of Mechanism: kept the security mechanisms relatively simple and straightforward by using well-established cryptographic libraries (cryptography package), standard protocols (ECDH, AES-GCM, HMAC-SHA256), and clear separation between authentication, encryption, and session management components rather than creating complex custom security implementations.

---

7. Conclusion

7.1 Summary of Achievements

ECDH key exchange protocol; Hybrid message encryption using ECDH + AES-GCM; Session management and expiration; Integrate and enhance existing user privileges system. All the above implied solutions are successful, following the best practice, and have almost no downgrade on system performance.

7.2 Security and Privacy Posture Assessment

Remaining Vulnerabilities:

For demonstration, the keys information is visible to admins and encryption/decryption is visible for users. They should NOT be visible in business environment.

Suggest an Attack: A spy admin can collect users' ECDH public keys and monitor session key derivation patterns through the inspection logs, then use this information combined with timing analysis or side-channel attacks to potentially

compromise future key exchanges or perform traffic analysis on encrypted communications.

7.3 Future Improvements

Improvement: May be a Visual interface to make testing and developing a bit easier.
