Course: ECE 572; Summer 2025

Instructor: Dr. Ardeshir Shojaeinasab

Student Name: Yujian Li

Student ID: V01046555

Assignment: Assignment 1

Date: June 20, 2025

GitHub Repository: https://github.com/YujianLiG208/ECE572 (sorry, I tried but unable to make it as private)

——————————————————————————

Executive Summary

In general, this assignment focuses on three things:

Identify the vulnerability in the provided SecureText application

Implement and demonstrate a realistic attack exploiting that vulnerability

Fix the vulnerability with a secure implementation

——————————————————————————

Table of Contents

——————————————————————————

1. Introduction

1.1 Objective

1.2 Scope

Task 1: Vulnerability Analysis (analyze the provided insecure messenger)

Task 2: Password Security (hashing, salting)

Task 3: Network Security (eavesdropping, message tampering, MAC attacks)

Note: Because copying the code from VSCode into a word document will make it difficult to read and the indentation will be difficult to display properly, all code snippets are screenshots.

1.3 Environment Setup

- Operating System: WIN 10, with WSL2 Ubuntu 24.04; KALI Virtual Machine
- Python Version: 3.13.0
- Key Libraries Used: argon2, secrets, base64, hashlib, pymd5, hmac
- Development Tools: Visual Studio Code with Copilot, Wireshark, VirtualBox, ChatGPT

————————————————————————

2. Task Implementation

2.1 Task 1: Security Vulnerability Analysis

2.1.1 Objective

Analyze the provided insecure messenger application and identify security weaknesses.

2.1.2 Implementation Details

I.

A. Vulnerability Analysis

1. **Description & Location**

```python
32     def load_users(self):
33         """Load users from JSON file or create empty dict if file doesn't exist"""
34         if os.path.exists(self.users_file):
35             try:
36                 with open(self.users_file, 'r') as f:
37                     return json.load(f)
38             except (json.JSONDecodeError, IOError):
39                 print(f"Warning: Could not load {self.users_file}, starting with empty user database")
40         return {}
41 
42     def save_users(self):
43         """Save users to JSON file with plaintext passwords (INSECURE!)"""
44         try:
45             with open(self.users_file, 'w') as f:
46                 json.dump(self.users, f, indent=2)
47         except IOError as e:
48             print(f"Error saving users: {e}")
49 
50     def create_account(self, username, password):
51         """Create new user account - stores password in PLAINTEXT!"""
52         if username in self.users:
53             return False, "Username already exists"
54 
55         # SECURITY VULNERABILITY: Storing password in plaintext!
56         self.users[username] = {
57             'password': password,  # PLAINTEXT PASSWORD!
58             'created_at': datetime.now().isoformat(),
59             'reset_question': 'What is your favorite color?',
60             'reset_answer': 'blue'  # Default for simplicity
61         }
62         self.save_users()
63         return True, "Account created successfully"
```

Line 32-63

- In create_account and save_users (lines 56–58 and 43–47), user passwords are written directly into a JSON file without any protection.

2. **Impact if Exploited**

- Any attacker who gains read access to the JSON immediately learns every user's password in cleartext—enabling full account takeover, credential stuffing on other services, and insider threats.

The following screenshot shows how simple the usernames and passwords being storage in the JSON file.

D: > UVIC > ECE572 > assignment > src > {} users.json > ...

```json
{
  "testuserA": {
    "password": "abcde",
    "created_at": "2025-06-18T19:30:07.550040",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue"
  },
  "testuserB": {
    "password": "67890",
    "created_at": "2025-06-18T19:31:25.501809",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue"
  }
}
```

3. **Relevant Security Principles**

- **Data Protection** (encrypt or hash sensitive data at rest)
- **Least Privilege** (restrict file access to only the service account)

- **Defense in Depth** (don't rely solely on filesystem permissions)

4. **Category**

Authentication / Data Protection

B. Attack Scenario

1. **Attacker Requirements**
   - Read access to the JSON file (e.g., via local compromise, misconfigured directory permissions, or LFI vulnerability).
2. **What They Could Achieve**
   - Steal every user's password, log in as any user, pivot to higher-privileged accounts, and use harvested credentials for attacks elsewhere.
3. **Final Thoughts & Mitigations**
   - Use a strong one-way hash instead of plaintext.
   - Encrypt the user store at rest or leverage OS-level encrypted volumes.
   - Tighten filesystem permissions so only the messenger process can read/write the file.
   - Implement secure password-reset flows.

II.

Line 70-79

1. **Description & Location**

Line 70-79

```
70            # SECURITY VULNERABILITY: Plaintext password comparison!
71            if self.users[username]['password'] == password:
72                return True, "Authentication successful"
73            else:
74                return False, "Invalid password"
75
76        def reset_password(self, username, new_password):
77            """Basic password reset - just requires existing username"""
78            if username not in self.users:
79                return False, "Username not found"
```

   - Plain comparison is used in the login routine, no hashing or salting is used, which means passwords are compared in cleartext.

- Unprotected reset (docstring of reset_password): the code only checks if username in self.users before allowing a password change, without verifying identity or secret.

2. **Impact if Exploited**
    - **Brute-force & timing attacks** become trivial: an attacker can rapidly guess passwords and detect correct characters via response timing.
    - **Account takeover via reset**: any unauthenticated caller who knows a valid username can reset that user's password and hijack the account.
3. **Relevant Security Principles**
    - **Data Protection**: use one-way hashing (e.g., bcrypt/Argon2) with per-user salt.
    - **Fail-Safe Defaults**: authentication should default to "deny" unless proof is provided.
    - **Least Privilege & Defense in Depth**: reset operations must require multi-factor verification (secret question, email token, etc.).
4. **Category**
    - **Authentication** (weak credential handling, no hashing, susceptible to timing attacks)
    - **Broken Access Control** (password reset without authorization)

B. Attack Scenarios

1. **Attacker Requirements**
    - Ability to invoke the messenger's login or reset API (e.g. via the console interface or by scripting calls if there's a network layer).
    - Knowledge of a target username (often trivial if usernames are public).
2. **What They Could Achieve**
    - **Login brute-force**: systematically try passwords; because no rate-limiting or hashing slows them, they discover valid credentials quickly.
    - **Timing side-channel**: infer correct password characters by measuring response times on equality checks.

- **Unauthorized reset**: call reset_password("victim", "newpass") and take over any account.

3. **Final Thoughts & Mitigations**

   - Replace plaintext storage/comparison with a vetted library (bcrypt/Argon2) and use a constant-time compare.
   - Implement proper password-reset workflow: verify ownership via email/SMS token or ask for the user's secret answer (and still compare that answer hashed).
   - Add rate-limiting, account lockout on repeated failures, and detailed auditing of reset events.

III.

1. **Description & Location**

Line 81-84

```
81          # SECURITY VULNERABILITY: No proper verification for password reset!
82          self.users[username]['password'] = new_password
83          self.save_users()
84          return True, "Password reset successful"
```

   - A hacker can easily reset any user's password after only checking that username exists—no further identity proof is required.

2. **Impact if Exploited**

   - Any unauthenticated attacker (or malicious insider) who can invoke the reset routine can arbitrarily reset **any** user's password and take over their account.

3. **Relevant Security Principles**

   - **Fail-Safe Defaults**: sensitive operations should deny by default unless explicitly authorized.
   - **Least Privilege**: password resets must require proof of identity (e.g., possession factor).
   - **Defense in Depth**: complement filesystem protections with application-level access controls.

4. **Category**

   - **Broken Access Control / Authentication Bypass**

B. Attack Scenario

1. **Attacker Requirements**

   - Ability to call the reset_password(username, new_password) function (e.g., via the console interface or exposed API).

   - Knowledge of a valid username (often trivial if usernames are guessable or public).

2. **What They Could Achieve**

   - Immediately reset the victim's password, log in as that user, access private messages or escalate further. If an admin account is targeted, full system compromise is possible.

3. **Final Thoughts & Mitigations**

   - **Enforce identity verification** before allowing a reset—e.g., send an email token or require answering a hashed secret question.

   - **Log and alert** on all reset attempts, and rate-limit/reset requests per account.

   - Store and compare secret answers securely (hashed), or better yet, adopt an industry-standard password-reset flow (time-limited tokens).

IV.

1. **Description & Location**

Line 23-30

```
23  class SecureTextServer:
24      def __init__(self, host='localhost', port=12345):
25          self.host = host
26          self.port = port
27          self.users_file = 'users.json'
28          self.users = self.load_users()
29          self.active_connections = {}  # username -> connection
30          self.server_socket = None
```

   - **No session tokens**: connections are tracked purely by username. There's no per-connection secret or token to prove identity once logged in.

   - **No transport security**: there's no SSL/TLS context or encryption applied to self.server_socket, so all messages flow in cleartext.

2. **Potential Impact**

- **Session Hijacking**: an attacker who knows a valid username can simply open a new connection claiming that username. That will overwrite the existing active_connections[username], effectively kicking the real user off and receiving all subsequent messages.
- **Eavesdropping & MITM**: because traffic is unencrypted, anyone on the same network (or in a position to intercept packets) can read every message exchanged.

3. **Relevant Security Principles**
    - **Least Privilege & Defense in Depth**: don't rely on "username alone" for session integrity; issue unguessable session tokens.
    - **Fail-Safe Defaults**: default to "deny" for any message if the session token is missing or invalid.
    - **Confidentiality of Data in Transit**: encrypt socket traffic (e.g., TLS) to protect against eavesdropping and tampering.
4. **Category**
    - **Session Management** (broken session integrity)
    - **Data Protection / Transport Security** (lack of encryption)

B. Attack Scenario

1. **Attacker Requirements**
    - Knowledge of a target's username (often guessable).
    - Network access to the server (local network or public Internet if host/port are exposed).
2. **Attack Steps & Impact**

Hijack Session

- Attacker connects, logs in as "victim" with any password (if earlier vulnerabilities let them bypass auth), or simply reuses the same username after guessing the password.
- The new connection entry in active_connections replaces the real user's socket.

        - Attacker now receives all messages intended for “victim” and can impersonate them on the chat.

Eavesdrop

        - Sniff network traffic using a tool like Wireshark.
        - Read every message—including passwords or reset tokens—since nothing is encrypted.

3. **Final Thoughts & Mitigations**
    - **Session Tokens**: upon successful login, generate a cryptographically random token and map active_connections[token] = connection; require that token for every subsequent message.
    - **Lock Concurrent Sessions**: either prevent multiple simultaneous logins or notify the first user if a second login occurs.
    - **Encrypt the Wire**: wrap the server socket in TLS (e.g., via Python’s ssl module), so all chat payloads cannot be read or forged in transit.
    - **Configuration Hardening**: bind by default to 127.0.0.1 and document that exposing on 0.0.0.0 carries risk.

V.

```python
202 class SecureTextClient:
203     def __init__(self, host='localhost', port=12345):
204         self.host = host
205         self.port = port
206         self.socket = None
207         self.logged_in = False
208         self.username = None
209         self.running = False
210 
211     def connect(self):
212         """Connect to the server"""
213         try:
214             self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
215             self.socket.connect((self.host, self.port))
216             return True
217         except ConnectionRefusedError:
218             print("Error: Could not connect to server. Make sure the server is running.")
219             return False
220         except Exception as e:
221             print(f"Connection error: {e}")
222             return False
223 
224     def send_command(self, command_data):
225         """Send command to server and get response"""
226         try:
227             self.socket.send(json.dumps(command_data).encode('utf-8'))
228             response = self.socket.recv(1024).decode('utf-8')
229             return json.loads(response)
230         except Exception as e:
231             print(f"Communication error: {e}")
232             return {'status': 'error', 'message': 'Communication failed'}
233 
234     def listen_for_messages(self):
235         """Listen for incoming messages in a separate thread"""
236         while self.running:
237             try:
238                 data = self.socket.recv(1024).decode('utf-8')
239                 if data:
240                     message = json.loads(data)
241                     if message.get('type') == 'MESSAGE':
242                         print(f"\n[{message['timestamp']}] {message['from']}: {message['content']}")
243                         print(">> ", end="", flush=True)
```

A. Vulnerability Analysis

1. **Description & Location**

    - **Unencrypted transport**: in connect() (lines 211–216) the client opens a plain TCP socket to host:port.

    - **Cleartext credentials & messages**: in send_command() (lines 226–231) it JSON-encodes whatever command_data holds—including login credentials—and sends them without any encryption.

- **Unsanitized output**: in listen_for_messages() (lines 236–243) it prints message['content'] directly to the terminal, opening the door to console-escape injection.

2. **Impact if Exploited**
    - **Eavesdropping**: any network-level sniffer (e.g. Wireshark, tcpdump) will see every password, token, and chat message in cleartext.
    - **MITM tampering**: an attacker performing ARP spoofing or DNS hijack can modify requests or forge server responses (e.g., inject malicious JSON or ANSI escape codes).
    - **Console injection**: a crafted message containing terminal-escape sequences could manipulate or wipe the user's terminal display, or even execute commands in some shells.
3. **Relevant Security Principles**
    - **Confidentiality of Data in Transit**: encrypt all traffic (e.g., TLS).
    - **Fail-Safe Defaults & Defense in Depth**: verify server identity and forbid fallback to plain TCP.
    - **Input Validation & Output Encoding**: never print untrusted content without sanitization.
4. **Category**
    - **Data Protection / Transport Security**
    - **Session Management** (no integrity checks or replay protection)
    - **Input/Output Handling** (lack of sanitization)

B. Attack Scenario

1. **Attacker Requirements**
    - Access to the same network segment as the client (e.g., public Wi-Fi or compromised LAN).
    - A packet-sniffing tool (Wireshark/tcpdump) .
2. **What They Could Achieve**
    - **Capture credentials** on every login attempt, then log in as any user.

- **Read all private conversations** in real time.
- **Inject or modify messages** on the fly—e.g., send a fake "reset_password" command or insert terminal escape codes that wipe the user's screen or trick them into entering sensitive data elsewhere.

3. **Final Thoughts & Mitigations**
    - **Upgrade to TLS:** wrap both client and server sockets in an SSL/TLS layer, verify certificates, and forbid insecure fallbacks.
    - **Introduce message integrity:** add HMAC or use an authenticated encryption mode.
    - **Sanitize outputs:** strip or encode control characters before printing messages to the terminal.
    - **Use a robust protocol library:** e.g., HTTP over TLS or a mature messaging framework that handles framing, encryption, and replay protection for you.

2.1.3 Challenges and Solutions
Not applicable for this section

2.1.4 Testing and Validation
Not applicable for this section

Test Cases

Not applicable for this section

——————————————————————————

2.2 Task 2: Securing Passwords at Rest

2.2.1 Objective
Implement secure password storage using hashing and salting techniques.

2.2.2 Implementation Details
Password Hashing Implementation

1. **Replace Plaintext Storage:**

As shown in the following screenshot, the create_account() method is modified to hash passwords before storing.

```python
    # NOTE: Passwords are now hashed using SHA-256 before storing
    def create_account(self, username, password):
        """Create new user account - stores password as SHA-256 hash"""
        if username in self.users:
            return False, "Username already exists"

        # Hash the password using SHA-256
        password_hash = hashlib.sha256(password.encode('utf-8')).hexdigest(

        self.users[username] = {
            'password': password_hash,  # SHA-256 HASHED PASSWORD
            'created_at': datetime.now().isoformat(),
            'reset_question': 'What is your favorite color?',
            'reset_answer': 'blue'  # Default for simplicity
        }
        self.save_users()
        return True, "Account created successfully"
```

The authenticate() method was also updated accordingly to compare hashed passwords.

```python
    def authenticate(self, username, password):
        """Authenticate user with hashed password comparison"""
        if username not in self.users:
            return False, "Username not found"

        # Hash the provided password and compare with stored hash
        password_hash = hashlib.sha256(password.encode('utf-8')).hexdigest()
        if self.users[username]['password'] == password_hash:
            return True, "Authentication successful"
        else:
            return False, "Invalid password"

```

After applying SHA-256, now the password are hashed, JSON file looks like this.

```json
{
  "testuserA": {
    "password": "36bbe50ed96841d10443bcb670d6554f0a34b761be67ec9c4a8ad2c0c44ca42c",
    "created_at": "2025-06-20T16:25:47.029465",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue"
  },
  "testuserB": {
    "password": "e2217d3e4e120c6a3372a1890f03e232b35ad659d71f7a62501a4ee204a3e66d",
    "created_at": "2025-06-20T16:26:04.293775",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue"
  }
}
```

Limitation of the SHA-256 fast hashing method:

- SHA-256 computes in <1 μs on modern CPUs and even faster on GPUs/ASICs, so attackers can try billions of guesses per second.
- Unless we rigorously implement per-user random salts and store them safely, you risk rainbow-table attacks or salt reuse.
- We can't easily adjust CPU/memory cost at runtime as threats evolve. However, slow hashing methods like Argon2 and bcrypt let us bump parameters without rewriting loop.

Therefore, SHA-256 is great for checksums, not for resisting password - cracking.

1. **Implement Slow Hashing**

Research which one is the best option(Assisted and summarized with AI):

1. PBKDF2:

It's a widely used algorithm, but considered weaker than the others in this list, especially against modern hardware attacks.

It uses a pseudorandom function (like HMAC) repeatedly with a salt to create a derived key.

It's relatively simple to implement and widely available, but its security is heavily reliant on the number of iterations used.

2. bcrypt:

It's a good, well-established algorithm, especially for legacy systems.

It uses the Blowfish cipher to create a password hash, making it computationally expensive to crack.

It has a built-in salt and iteration count, simplifying its use.

3. scrypt:

It's designed to be memory-hard, meaning it requires a significant amount of memory to compute, making it resistant to hardware-based attacks.

It has several parameters that can be adjusted, including the memory cost, the number of iterations, and the degree of parallelism.

It can be more resource-intensive than bcrypt, but offers stronger protection against certain types of attacks.

4. Argon2:

It's the newest and generally considered the most secure of the four.

It offers configurable time and memory usage, making it adaptable to different systems and security needs.

It's designed to be resistant to various attacks, including those targeting GPUs and side-channel vulnerabilities.

Therefore, I decide to choose Argon2 and imply it. For here, a different copy of securetext.py is created.

```python
    def create_account(self, username, password):
        """Create new user account - stores password using Argon2 hash"""
        if username in self.users:
            return False, "Username already exists"

        ph = PasswordHasher()
        hashed_password = ph.hash(password)
        self.users[username] = {
            'password': hashed_password,  # Argon2 hash
            'created_at': datetime.now().isoformat(),
            'reset_question': 'What is your favorite color?',
            'reset_answer': 'blue'  # Default for simplicity
        }
        self.save_users()
        return True, "Account created successfully"

    def authenticate(self, username, password):
        """Authenticate user with Argon2 password verification"""
        if username not in self.users:
            return False, "Username not found"

        ph = PasswordHasher()
        try:
            if ph.verify(self.users[username]['password'], password):
                return True, "Authentication successful"
            else:
                return False, "Invalid password"
        except argon2_exceptions.VerifyMismatchError:
            return False, "Invalid password"
        except Exception as e:
            return False, f"Authentication error: {e}"
```

Among all three variants of Argon2, Argon2d's strength is the resistance against time–memory trade-offs, while Argon2i's focus is on resistance against side-channel attacks. Accordingly, Argon2i was originally considered the correct choice for password hashing and password-based key derivation. In practice it turned out that a combination of d and i – that combines their strengths – is the better choice https://argon2-cffi.readthedocs.io/en/stable/argon2.html

Therefore, I pick it as my choice to imply here.

Salt Implementation

1. **Add Salt Generation**

```python
def create_account(self, username, password):
    """Create new user account - stores password using Argon2 hash and unique salt"""
    if username in self.users:
        return False, "Username already exists"

    # Generate a unique random 128-bit salt for this user
    salt_bytes = secrets.token_bytes(16)  # 128 bits
    salt_b64 = base64.b64encode(salt_bytes).decode('utf-8')

    ph = PasswordHasher()
    # Combine password and salt for hashing
    password_with_salt = password + salt_b64
    hashed_password = ph.hash(password_with_salt)
    self.users[username] = {
        'password': hashed_password,  # Argon2 hash
        'salt': salt_b64,
        'created_at': datetime.now().isoformat(),
        'reset_question': 'What is your favorite color?',
        'reset_answer': 'blue'  # Default for simplicity
    }
    self.save_users()
    return True, "Account created successfully"
```

```python
def authenticate(self, username, password):
    """Authenticate user with Argon2 password verification and stored salt"""
    if username not in self.users:
        return False, "Username not found"

    ph = PasswordHasher()
    try:
        salt_b64 = self.users[username].get('salt')
        if not salt_b64:
            return False, "Salt missing for user"
        password_with_salt = password + salt_b64
        if ph.verify(self.users[username]['password'], password_with_salt):
            return True, "Authentication successful"
        else:
            return False, "Invalid password"
    except argon2_exceptions.VerifyMismatchError:
        return False, "Invalid password"
    except Exception as e:
        return False, f"Authentication error: {e}"
```

Generate a unique random salt for each user (minimum 128 bits), store the salt alongside the hashed password, and authentication( ) is modified to use the stored salt.

2. Migration

Migrate existing plaintext passwords, ensure backward compatibility during the transition

```python
        # --- Password migration for backward compatibility ---
        # If the stored password is not an Argon2 hash, treat it as plaintext and migrate
        if not (isinstance(stored_password, str) and stored_password.startswith("$argon2")):
            # Legacy plaintext password: compare directly
            if stored_password == password:
                # Migrate: generate salt, hash password, update user record
                if not salt_b64:
                    salt_bytes = secrets.token_bytes(16)
                    salt_b64 = base64.b64encode(salt_bytes).decode('utf-8')
                    self.users[username]['salt'] = salt_b64
                password_with_salt = password + salt_b64
                hashed_password = ph.hash(password_with_salt)
                self.users[username]['password'] = hashed_password
                self.save_users()
                return True, "Authentication successful (password migrated)"
            else:
                return False, "Invalid password"
        # --- End migration logic ---

        try:
            if not salt_b64:
                return False, "Salt missing for user"
            password_with_salt = password + salt_b64
            if ph.verify(self.users[username]['password'], password_with_salt):
                return True, "Authentication successful"
            else:
                return False, "Invalid password"
        except argon2_exceptions.VerifyMismatchError:
            return False, "Invalid password"
        except Exception as e:
            return False, f"Authentication error: {e}"
```

2.2.3 Challenges and Solutions

When simulating attack, I need to transfer files from Windows Host to KALI VM.

Solution: On my host, In the folder containing JSONs and the dict_attack.py, use powershell to run a simple HTTP server:

python3 -m http.server 8000

In KALI VM, download them with curl or wget:

wget http://<HOST_IP>:8000/users_unsalted.json

wget http://<HOST_IP>:8000/users_salted.json

wget http://<HOST_IP>:8000/ dict_attack.py

Close the powershell HTTP server after all transaction are finished.

2.2.4 Testing and Validation

D: > UVIC > ECE572 > assignment > src > {} users.json > ...


```json
{
  "testuserA": {
    "password": "$argon2id$v=19$m=65536,t=3,p=4$nJILtKzpEwmktiXOYrQqxg$3JNf3Eflh8XDgV7aRU0W1anDvFDvOqfbEcOvk6Pxv8Q",
    "created_at": "2025-06-18T19:30:07.550040",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue",
    "salt": "rmV2UW1d0odEQbrHowmapA=="
  },
  "testuserB": {
    "password": "$argon2id$v=19$m=65536,t=3,p=4$DtpJ3m7NsL0y5jc9OJ31vA$ygKb9XpFEY7qN1sT8rCitUpjMPPCv4pGBNQkHNlYzvg",
    "created_at": "2025-06-18T19:31:25.501809",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue",
    "salt": "4jcW2npKusMOYdXw5llaVA=="
  }
}
```


As we can see in the JSON file, after sign in, the two existing testuser account passwords are now storage as hashed and salted.

(For the attack, please refer to the corresponding dictionary attack in part 4)

——————————————————————————

2.3 Task 3: Network Security and MAC Implementation

2.3.1 Objective

Demonstrate network attacks and implement (flawed and secure) message authentication.

2.3.2 Implementation Details

How could plain text communication being taken advantage of ?

First, the attacker poisons ARP tables of both client and server—using Cain & Abel on Windows (or arpspoof on Linux)—so that all packets between the two flow through the attacker's machine. With traffic now diverted, they fire up Wireshark (or tcpdump) to confirm they're seeing the JSON-formatted chat:

Next, they switch from passive sniffing to active tampering. On Windows they might load the WinDivert driver and run a small Python/Scapy script that captures TCP packets on port 12345, parses the JSON payload, rewrites fields (for example changing "from":"alice" to "from":"attacker" or injecting a "reset_password" command), then recalculates TCP/IP checksums and forwards the modified packet on to the server.

Because SecureText lacks TLS and message authentication, the server and client blindly accept these forged messages. The attacker can thus impersonate users, tamper with conversations, or exfiltrate credentials—demonstrating why you need end-to-end encryption plus HMAC or certificates to prevent any unauthorized modification. (Summarized with help of ChatGPT)

```python
    # --- Begin MAC and Key Management additions ---

    def get_shared_key(self):
        """
        Returns the pre-shared key for all users.
        Insecure: In a real system, use per-user keys and secure key exchange.
        """
        # Hardcoded pre-shared key (for demo only)
        return "SuperSecretSharedKey123"

    def mac(self, key, message):
        """
        Compute a flawed MAC using MD5(key || message).
        Args:
            key (str): The shared secret key.
            message (str): The message to authenticate.
        Returns:
            str: Hex digest of the MAC.
        """
        data = (key + message).encode('utf-8')
        return hashlib.md5(data).hexdigest()

    def verify_mac(self, key, message, mac_value):
        """
        Verify the MAC for a given message.
        Returns True if valid, False otherwise.
        """
        expected_mac = self.mac(key, message)
        return expected_mac == mac_value

    # --- End MAC and Key Management additions ---
```

Implementing a flawed H(k||m) MAC

```python
# --- Begin Merkle-Damgård length extension attack demo ---

def forge_mac_length_extension(self, orig_message, orig_mac, append_data):
    """
    Demonstrate Merkle-Damgård length extension attack on MAC(k||m) = MD5(k||m).
    Given orig_message and its MAC, forge a valid MAC for orig_message||padding||append_data.
    Returns (forged_message, forged_mac).
    """

    # Helper: MD5 padding for a message of given length (in bytes)
    def md5_padding(msg_len_bytes):
        # MD5 uses 64-byte blocks
        pad = b'\x80'
        pad += b'\x00' * ((56 - (msg_len_bytes + 1) % 64) % 64)
        pad += struct.pack('<Q', msg_len_bytes * 8)
        return pad

    # The attacker doesn't know the key, but can guess its length (e.g., 16 bytes)
    key_len_guess = 16
    total_len = key_len_guess + len(orig_message)
    padding = md5_padding(total_len)
    forged_message = orig_message.encode('utf-8') + padding + append_data.encode('utf-8')

    # Parse original MAC as MD5 state

    # Use a 3rd-party library to set MD5 state (Python's hashlib doesn't support this natively)
    try:
        # Recompute padding using pymd5 for compatibility
        pymd5_pad = pymd5_padding((key_len_guess + len(orig_message)) * 8)
        m = md5(state=bytes.fromhex(orig_mac), count=(key_len_guess + len(orig_message) + len(pymd5_pad)) * 8)
        m.update(append_data)
        forged_mac = m.hexdigest()
        return forged_message, forged_mac
    except ImportError:
        # If pymd5 is not available, just return None
        print("pymd5 required for length extension attack demo (pip install pymd5)")
        return None, None

# Example usage for demonstration:
# orig_msg = "CMD=SET_QUOTA&USER=bob&LIMIT=100"
# orig_mac = self.mac(self.get_shared_key(), orig_msg)
# forged_msg, forged_mac = self.forge_mac_length_extension(orig_msg, orig_mac, "&LIMIT=1000000")
# Now forged_msg and forged_mac can be sent to the server, which will accept them as valid!

# --- End Merkle-Damgård length extension attack demo ---
```

Implementing a secure MAC (HMAC-SHA256)

```python
    # --- Begin Secure MAC (HMAC-SHA256) implementation ---
    # HMAC-SHA256 is secure because it uses a secret key and the SHA-256 hash function in a specific construction
    # that prevents common attacks (like length extension). Only someone with the key can compute or verify the MAC,
    # ensuring both message integrity and authentication. HMAC's design has been extensively analyzed and is widely
    # trusted in cryptographic applications.

    def mac(self, key, message):
        """
        Compute a secure MAC using HMAC-SHA256.
        Args:
            key (str): The shared secret key.
            message (str): The message to authenticate.
        Returns:
            str: Hex digest of the MAC.
        """
        key_bytes = key.encode('utf-8')
        msg_bytes = message.encode('utf-8')
        return hmac.new(key_bytes, msg_bytes, hashlib.sha256).hexdigest()

    def verify_mac(self, key, message, mac_value):
        """
        Verify the MAC for a given message using HMAC-SHA256.
        Returns True if valid, False otherwise.
        """
        expected_mac = self.mac(key, message)
        # Use hmac.compare_digest for timing-attack resistance
        return hmac.compare_digest(expected_mac, mac_value)

    # --- End Secure MAC (HMAC-SHA256) implementation ---
```

2.3.3 Challenges and Solutions

HASHBUMP setting failed, which means no solution to do length extension

2.3.4 Testing and Validation

Please refer to the corresponding attack in the attack section

——————————————————————————

3. Security Analysis

3.1 Vulnerability Assessment

Vulnerability	Severity	Impact	Location	Mitigation
User password storage in plain text	high	Any attacker have access to JSON file can get the password	32-63	Apply slow hashing and salt
Plain comparison is used in the login routine	high	Easy target for Brute-force & timing attacks	70-79	use one-way hashing (e.g., bcrypt/Argon2) with per-user salt.
No identity proof	high	Once the password is compromised, the attacker	81-84	2^{nd} authentication

		have access		
No session tokens	high	Man-in the middle attack	23-30	transport layer security
opens a plain TCP socket	high	Eavesdropping	211-216	Encrypt communication

3.2 Security Improvements

Before vs. After Analysis:

- **Data Protection**: Slow hashing and salt applied, make the attacker extremely difficult to gain passwords.
- **Communication Security**: HMAC-SHA256 and MAC implied, reduce the man-in-the-middle attack

3.3 Threat Model

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

Threat Actors:

9. **Passive Network Attacker**: Can intercept but not modify traffic
10. **Active Network Attacker**: Can intercept and modify traffic
11. **Malicious Server Operator**: Has access to server and database
12. **Compromised Client**: Attacker has access to user's device

Security Properties Achieved:

- ~~[ ] Confidentiality~~
- ~~[ ] Integrity~~
- [ ] Authentication
- [ ] Authorization
- [ ] Non-repudiation
- ~~[ ] Perfect Forward Secrecy~~
- [ ] Privacy

———————————————————————————

4. Attack Demonstrations

4.1 Attack 1: Dictionary Attack

4.1.1 Objective

If the attacker can have access to the Password storage JSON file, they will try to get the password, comparing fast hashing vs salted slow hashing performance when facing such attack.

4.1.2 Attack Setup

KALI VM and the rockyou dictionary.

move the two JSONs and the dictionary attack python script to KALI.

4.1.3 Attack Execution

To simulate a dictionary attack example and make comparison, I have create a new attack python script dict_attack.py with assistance of Copilot. (see deliverables)

Unsalted JSON with SHA256 fast hashing was breached almost immediately.

However, I got nothing after waiting for 30 mins to see if the same attack can breach salted JSON with Argon2id slow hashing. My host physical laptop's fan was very loud, and the case is hot.

4.1.4 Results and Evidence

```
┌──(kali㉿KALI)-[~]
└─$ time /home/kali/dict_attack.py -u users_unsalted.json -w /usr/share/wordlists/rockyou.
txt
Unsalted attack: {'testuserA': 'abcde', 'testuserB': '67890'}

real    28.41s
user    28.28s
sys     0.14s
cpu     100%

┌──(kali㉿KALI)-[~]
└─$ time /home/kali/dict_attack.py -s users_salted.json -w /usr/share/wordlists/rockyou.tx
t
█
```

4.1.5 Mitigation

slow hashing with salt could greatly increase the time and computing power cost of any potential attackers.

————————————————————————————

4.2 Attack 2: Eavesdropping

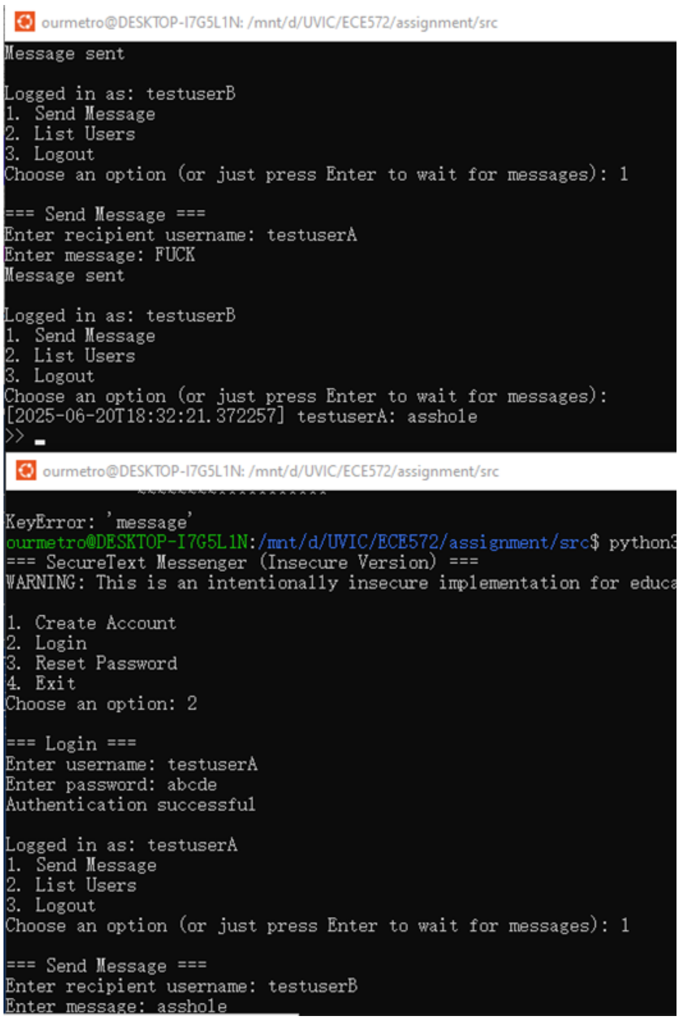
4.2.1 Objective

The original securetext app use plain text communication, which is an easy target.

4.2.2 Attack Setup

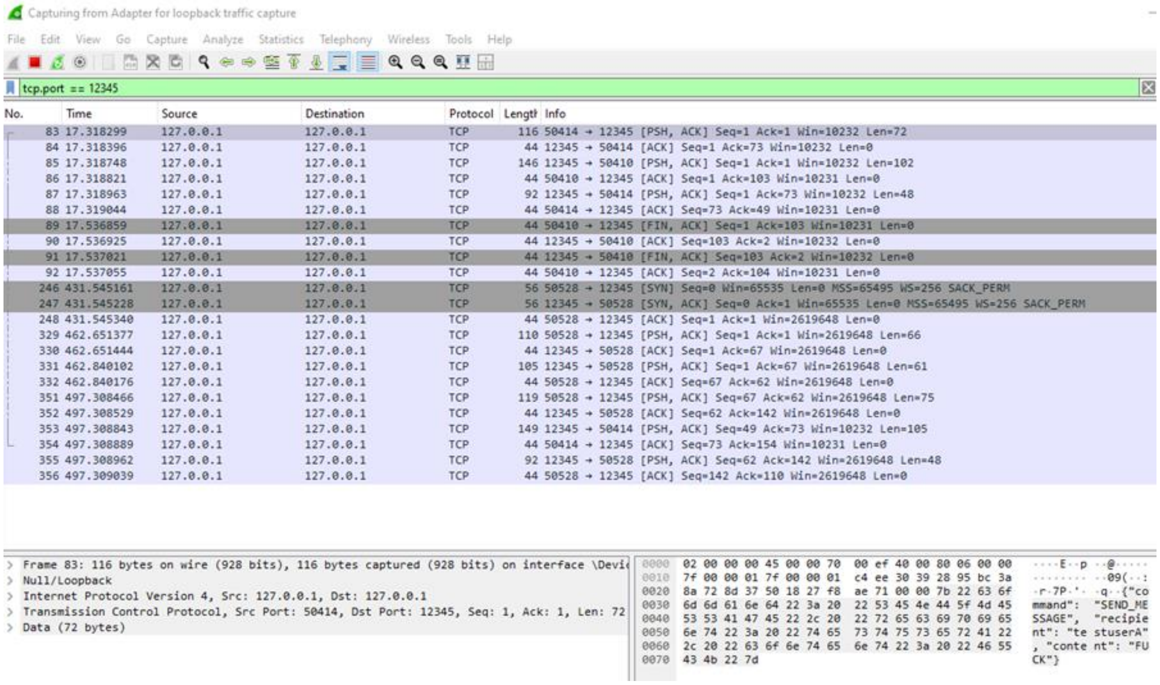
Set up the Wireshark filter rule tcp.port == 12345, as SecureText server listens on port 12345 by default.

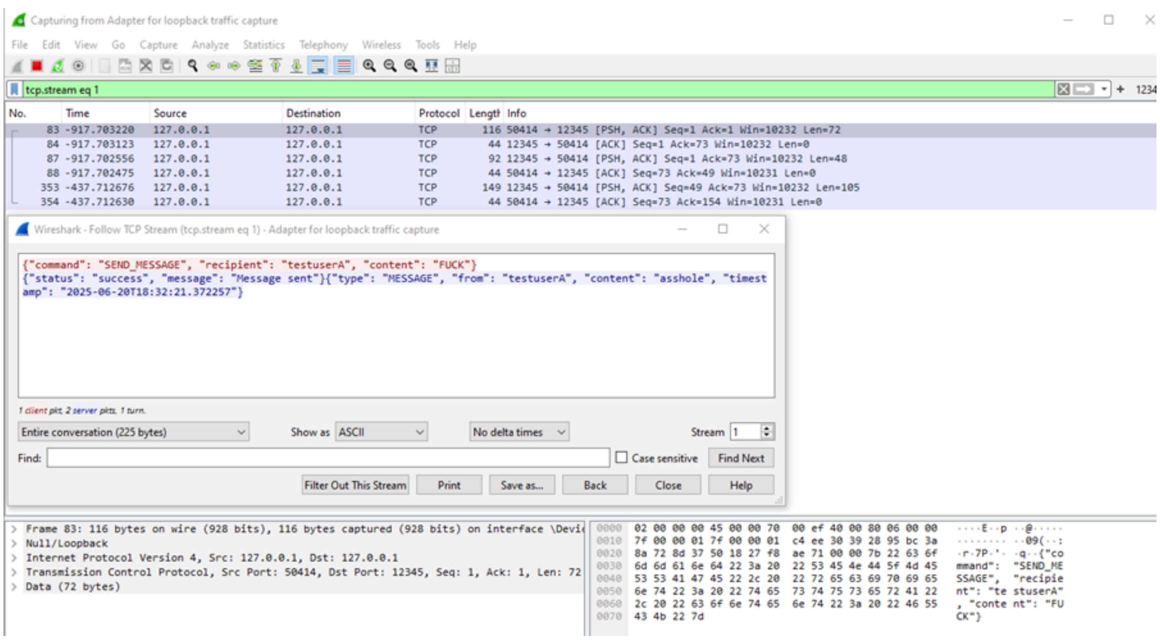
4.2.3 Attack Execution



I signed in as both test users, sending message to each other, and track the packages being captured on wire shark.

4.2.4 Results and Evidence





For detailed Result, refer to the pdf file in the deliverable pdf.

4.2.5 Mitigation

Apply HMAC-SHA256

———————————————————————————

5. Performance Evaluation

Basic test results in terms of resources used in terms of hardware and time. Also, if the test has limitations and fix worked properly(test passed or failed)

Not applicable for this one

6. Lessons Learned

6.1 Technical Insights

6.2 Security Principles

———————————————————————————

7. Conclusion

7.1 Summary of Achievements

Communication privacy improved, password storage improved.

7.2 Security and Privacy Posture Assessment

Actually not perfectly secure, but being improved from the original.

Remaining Vulnerabilities:

- Vulnerability 1: password reset still not being improved
- Vulnerability 2: sign in just require password.

Suggest an Attack: In two lines mention a possible existing attack to your current version in abstract

7.3 Future Improvements

13. **Improvement 1**: authentication on password reset, 2nd factor authentication apply.
14. **Improvement 2**: complete the length extension attack, for this time due to system environment setup, I did not have time to do that.

———————————————————————————