A close up of a logo

Description automatically generatedA blue and white logo with a ribbon

Description automatically generated

**AIN SHAMS UNIVERSITY**

**FACULTY OF ENGINEERING**

**CREDIT HOURS ENG. PROGRAM**

**Computer engineering and software systems**

**Ain Shams University**

**Faculty of Engineering**

**Fall 24 || Semester 1**

**Project**

**Secure Communication Suit**

**Computer and Network Security**

**CSE 451**

**Submitted to:**

**Prof. Ayman Bahaa El Din**

**Prof. Islam Tharwat Abdel Halim**

**Eng. Hesham Fathy Abdel Razik**

**Submitted by:**

|  |  |
| --- | --- |
| **Mohamed Hesham El Said Zidan** | **20P7579** |
| **Ahmed Seif Elsayed Ibrahim** | **20P7668** |
| **Sherwet Mohamed Khalil Barkat** | **20P8105** |
| **Yousef Fayez Ibrahim** | **2101616** |

Contents

[Phase 1 3](#_Toc185967439)

[Overview 3](#_Toc185967440)

[Project Plan 3](#_Toc185967441)

[Software Requirements 4](#_Toc185967442)

[Functional Requirements 4](#_Toc185967443)

[Non-Functional Requirements 5](#_Toc185967444)

[Design Documents 6](#_Toc185967445)

[Use case Diagram 6](#_Toc185967446)

[Phase 2 7](#_Toc185967447)

[Development of Cryptographic Modules 7](#_Toc185967448)

[Block Cipher 7](#_Toc185967449)

[Public key cryptosystem 8](#_Toc185967450)

[Hashing 13](#_Toc185967451)

[Phase 3 15](#_Toc185967452)

[Key management Moule 15](#_Toc185967453)

[Authentication Module 18](#_Toc185967454)

[Phase 4 19](#_Toc185967455)

[Integration of the Secure Communication Suite 19](#_Toc185967456)

[Real Tools or Services similar to our suite: 33](#_Toc185967457)

[Cost Analysis for a prototype 35](#_Toc185967458)

[Business Value to the Market 38](#_Toc185967459)

# 

# Phase 1

## Overview

This project aims to develop a **Secure Communication Suite** in Python, a comprehensive application that integrates various cryptographic techniques and security protocols. The suite will feature **block ciphers** for symmetric encryption, **public key cryptosystems** for asymmetric encryption, and **hashing functions** for data integrity. It will also incorporate **key management** solutions for secure key distribution and storage, and **authentication** mechanisms to verify user identities. The application will be designed to secure **internet services**, protecting data in transit and at rest.

**Features and Specifications:**

**Block Cipher Module:** Implement AES for symmetric encryption.

**Public Key Cryptosystem Module:** Implement RSA for asymmetric encryption.

**Hashing Module:** Implement SHA-256 for data integrity checks.

**Key Management Module:** Develop secure methods for key generation, distribution, and storage.

**Authentication Module:** Implement certificate-based authentication mechanisms.

**Internet Services Security Module:** Apply the cryptographic modules to secure data for internet services.

## Project Plan

**1. System Architecture:**

* **Client-Server Architecture:** A client-server architecture will be implemented, where the client application will interact with the server-side application for encryption, decryption, and key management.
* **Modular Design:** The system will be divided into modules for encryption/decryption, key management, authentication, and user interface.
* **Data Flow:** A clear data flow diagram will be created to visualize the interaction between different components.

**2. Data Structures:**

* **Key Storage:** Securely store public and private keys using appropriate cryptographic techniques.
* **User Information:** Store user information, including usernames, passwords, and public keys.
* **Encrypted Data:** Store encrypted data in a secure format.

**3. Algorithm Selection:**

* **Symmetric Encryption:** Implement AES with various modes of operation (CBC chosen).
* **Asymmetric Encryption:** Implement RSA for key exchange and digital signatures.
* **Hashing:** Implement SHA-256 for message integrity and password hashing.

**4. Security Considerations:**

* **Secure Coding Practices:** Adhere to secure coding principles to prevent vulnerabilities.
* **Input Validation:** Validate user input to avoid injection attacks.
* **Secure Key Management:** Implement robust key generation, storage, and distribution mechanisms.
* **Regular Security Audits:** Conduct regular security audits to identify and address vulnerabilities.

**5. User Interface:**

* Implement Simple Terminal interface

**6. Testing and Deployment:**

* **Unit Testing:** Test individual components to ensure correct functionality.
* **Integration Testing:** Test the integration of different modules.
* **Security Testing:** Conduct security testing to identify vulnerabilities.

## Software Requirements

### Functional Requirements

1. **Encryption/Decryption:**
   * Implement symmetric encryption algorithms (e.g., AES) for confidentiality.
   * Implement asymmetric encryption algorithms (RSA) for key exchange and digital signatures.
   * Implement hashing algorithms (e.g., SHA-256, MD5) for data integrity verification.
2. **Key Management:**
   * Generate, store, and manage cryptographic keys securely.
   * Implement key exchange protocols for secure key distribution.
3. **Authentication:**
   * Implement password-based and certificate-based authentication using digital certificates.
4. **Digital Signatures:**
   * Generate and verify digital signatures for message authenticity and integrity.
5. **Secure Communication Protocols:**
   * Implement secure protocols for file transfer and remote access.
6. **User Interface:**
   * Provide a user-friendly interface using command Line interpreter for Registration, Login, key management, encryption/decryption and secure communication.
   * Allow users to easily import and export keys.
   * Provide clear instructions and error messages.

### Non-Functional Requirements

1. **Security:**
   * Prioritize security by using strong cryptographic algorithms and secure coding practices.
   * Implement robust access control mechanisms.
   * Protect against common attacks (e.g., man-in-the-middle, brute-force, phishing).
2. **Performance:**
   * Ensure efficient encryption/decryption operations.
   * Optimize the application for performance.
   * Handle large file sizes and high data throughput.
3. **Usability:**
   * Design a user-friendly interface.
   * Provide clear instructions and documentation.
   * Minimize user effort for common tasks.
4. **Reliability:**
   * Implement error handling and recovery mechanisms.
   * Ensure data integrity and consistency.
   * Provide regular updates and security patches.
5. **Scalability:**
   * Design the system to handle increasing workloads and user base.
   * Consider distributed architectures for scalability.
6. **Maintainability:**
   * Write clean, well-structured, and well-documented code.
   * Use modular design principles.

## Design Documents

### Use case Diagram

A diagram of a person with circles and text

Description automatically generated

# Phase 2

## Development of Cryptographic Modules

### Block Cipher

As for the Block Cipher we used AES

A computer screen shot of text

Description automatically generated

A black background with white text

Description automatically generated

A computer code with blue and orange text

Description automatically generated

**Introduction**

This document describes the AESHandler class, a Python class designed for encrypting and decrypting data using the Advanced Encryption Standard (AES) algorithm in Cipher Block Chaining (CBC) mode. It utilizes the pycryptodome library for cryptographic operations.

**Dependencies**

The AESHandler class relies on the following external libraries:

* pycryptodome: A Python library providing cryptographic functionalities.

**Attributes**

* key (bytes): The encryption key used for AES operations. It should be a byte string of a valid AES key length (16 bytes).
* iv (bytes): The initialization vector (IV) used in CBC mode. It should be a random byte string of the AES block size (16 bytes).
* cipher (AES.Cipher): An instance of the AES cipher object created with the provided key and IV in CBC mode.

**Methods**

* \_\_init\_\_(self, key, iv): The constructor initializes an AESHandler object, storing the key, IV, and creating the cipher object.
* encrypt(self, plain\_text: str) -> bytes: Encrypts a plain text string using AES CBC mode. It returns the encrypted ciphertext as binary data (bytes).
  + Pads the plain text to a multiple of the AES block size (16 bytes).
  + Encrypts the padded plain text using the cipher object.
* decrypt(self, cipher\_text: bytes) -> str: Decrypts a ciphertext using AES CBC mode. It returns the decrypted plain text as a string.
  + Decrypts the ciphertext using the cipher object.
  + Removes the padding applied during encryption.
  + Decodes the unpadded bytes back to a string.

### Public key cryptosystem

A computer screen shot of text

Description automatically generated

This Python function generates a pair of RSA public and private keys using the cryptography library. RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm that uses a pair of keys: a public key for encryption and a private key for decryption. 1

**Parameters:**

None.

**Returns:**

A tuple containing:

1. **Public Key:** An RSA public key object.
2. **Private Key:** An RSA private key object.

**Explanation:**

1. **Import Necessary Modules:**
   * **cryptography.hazmat.primitives.serialization:** This module provides functions for serializing and deserializing cryptographic keys.
   * **cryptography.hazmat.primitives.asymmetric.rsa**: This module provides functions for generating and using RSA keys.
2. **Generate Private Key:**
   * **private\_key = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048):** This line generates a new RSA private key. The public\_exponent is a commonly used value for RSA keys, and the key\_size specifies the key length in bits (in this case, 2048 bits).
3. **Extract Public Key:**
   * **public\_key = private\_key.public\_key():** This line extracts the public key from the private key. The public key can be shared with others to allow them to encrypt data that only the holder of the private key can decrypt.
4. **Return Key Pair:**
   * The function returns a tuple containing the public key and the private key.

A computer screen shot of text

Description automatically generated

A computer screen shot of text

Description automatically generated

A computer screen shot of text

Description automatically generated

A computer screen shot of a program

Description automatically generated

**Overview**

The **RSAHandler** class provides methods for encrypting, decrypting, signing, and verifying data using RSA cryptography. It leverages the cryptography library for cryptographic operations.

**Class Attributes**

* **public\_key:** An RSA public key object used for encryption and verification.
* **private\_key:** An RSA private key object used for decryption and signing.

**Methods**

**1. \_\_init\_\_(self, public\_key=None, private\_key=None)**

* **Purpose:** Initializes an **RSAHandler** instance with optional public and private keys.
* **Parameters:**
  + **public\_key**: An RSA public key object.
  + **private\_key**:
  + An RSA private key object.
* **Description:**
  + Stores the provided public and private keys as class attributes.

**2. encrypt(self, data)**

* **Purpose:** Encrypts data using the public key.
* **Parameters:**
  + data: The data to be encrypted.
* **Returns:**
  + The encrypted data.
* **Description:**
  + Converts the data to bytes.
  + Encrypts the data using the public key with OAEP padding and SHA-256 as the hash algorithm.
  + Returns the encrypted ciphertext.

**3. decrypt(self, data)**

* **Purpose:** Decrypts data using the private key.
* **Parameters:**
  + data: The encrypted data.
* **Returns:**
  + The decrypted data, or None if decryption fails.
* **Description:**
  + Decrypts the data using the private key with OAEP padding and SHA-256 as the hash algorithm.
  + Handles potential exceptions during decryption and returns None on failure.

**4. sign(self, message)**

* **Purpose:** Signs a message using the private key.
* **Parameters:**
  + message: The message to be signed.
* **Returns:**
  + The signature of the message.
* **Description:**
  + Converts the message to bytes.
  + Signs the message using the private key with PKCS#1 v1.5 padding and SHA-256 as the hash algorithm.
  + Returns the signature.

**5. verify(self, message, signature)**

* **Purpose:** Verifies the signature of a message using the public key.
* **Parameters:**
  + message: The message to be verified.
  + signature: The signature of the message.
* **Returns:**
  + True if the signature is valid, False otherwise.
* **Description:**
  + Verifies the signature using the public key with PKCS#1 v1.5 padding and SHA-256 as the hash algorithm.
  + Returns True if the signature is valid, otherwise returns False.

### Hashing

A computer screen shot of a code

Description automatically generated

**Purpose:**

This Python code snippet implements a function to generate a SHA-256 hash of a given input data. SHA-256 is a cryptographic hash function that produces a 256-bit hash value.

1. **Import the hashlib Module:**

* The **hashlib** module provides various cryptographic hash functions, including SHA-256.

2. **Create a SHA-256 Hash Object:**

* **sha256\_hash = hashlib.sha256():** Creates a new SHA-256 hash object.

3. **Update the Hash Object with Data:**

* **sha256\_hash.update(data.encode()):** Updates the hash object with the input data. Before updating, the data is converted to bytes using the **encode()** method. This ensures that the hashing algorithm processes binary data.

4. **Get the Hexadecimal Digest:**

* **sha256\_hash.hexdigest():** Returns the final hash value in hexadecimal format.

A computer screen shot of text

Description automatically generated

**Purpose:**

This Python code snippet implements a function to generate an MD5 hash of a given input data. MD5 is a cryptographic hash function, though it's considered less secure for modern cryptographic applications due to known vulnerabilities. It's primarily used for legacy systems or simple checksum calculations.

1. **Import the hashlib Module:**

* The hashlib module provides various cryptographic hash functions, including MD5.

2. **Create an MD5 Hash Object:**

* md5\_hash = hashlib.md5(): Creates a new MD5 hash object.

3. **Update the Hash Object with Data:**

* md5\_hash.update(data.encode()): Updates the hash object with the input data. Before updating, the data is converted to bytes using the encode() method. This ensures that the hashing algorithm processes binary data.

4. **Get the Hexadecimal Digest:**

* md5\_hash.hexdigest(): Returns the final hash value in hexadecimal format.

A screen shot of a computer code

Description automatically generated

**Purpose:**

This Python function verifies the integrity of data by comparing its computed hash with a given hash value. It supports two hashing algorithms: SHA-256 and MD5.

**Parameters:**

* **data**: The data to be hashed and compared.
* **given\_hash**: The expected hash value of the data.
* **algorithm**: The hashing algorithm to use, either "sha256" or "md5". Defaults to "sha256".

**Returns:**

* **True**: If the computed hash matches the given hash.
* **False**: If the computed hash does not match the given hash.

# Phase 3

## Key management Moule

A screen shot of a computer code

Description automatically generated

A computer screen shot of text

Description automatically generated

**1. Key Generation Functions**

**1.1 generate\_aes\_key(key\_size=16)**

* **Purpose:** Generates a random AES (Advanced Encryption Standard) key for use in symmetric encryption algorithms.
* **Parameters:**
  + key\_size (optional, default 16): The desired length of the key in bytes. Common values are 16, 24, and 32.
* **Returns:** A random byte string representing the generated AES key.

**1.2 generate\_iv(block\_size=16)**

* **Purpose:** Generates a random initialization vector (IV) for use in certain block cipher modes of operation.
* **Parameters:**
  + block\_size (optional, default 16): The block size of the cipher mode that will use the IV.
* **Returns:** A random byte string representing the generated IV.

**1.3 generate\_rsa\_keys()**

* **Purpose:** Generates a new RSA (Rivest–Shamir–Adleman) public-private key pair using the cryptography library.
* **Parameters:** None
* **Returns:**
  + A tuple containing two elements:
    - The generated public key as a rsa.PublicKey object.
    - The generated private key as a rsa.PrivateKey object.

**2. RSA Key Management Functions**

**2.1 generate\_and\_store\_rsa\_keys(username)**

* **Purpose:** Generates a new RSA key pair for the specified user and stores them in dedicated files within the user's directory.
* **Parameters:**
  + username: A string representing the username for whom keys are generated.
* **Returns:** None (prints messages to the console)
* **Functionality:**
  + Checks for existing key files for the user.
  + If keys don't exist, generates a new RSA key pair using rsa.newkeys.
  + Saves the public key to data/keys/{username}\_public.pem.
  + Saves the private key to data/keys/{username}\_private.pem.
  + Prints success or existing key messages.

**2.2 load\_rsa\_keys(username)**

* **Purpose:** Loads an existing RSA public-private key pair for the specified user from their dedicated files.
* **Parameters:**
  + username: A string representing the username for whom keys are loaded.
* **Returns:** A tuple containing two elements:
  + The loaded public key as a rsa.PublicKey object.
  + The loaded private key as a rsa.PrivateKey object.
* **Functionality:**
  + Opens the user's private and public key files stored in data/keys.
  + Loads the keys using rsa.PrivateKey.load\_pkcs1 and rsa.PublicKey.load\_pkcs1 respectively.
  + Returns the loaded key pair.

## Authentication Module

A computer screen with text

Description automatically generated

**Function: authenticate\_user**

**Purpose:** This function authenticates a user by validating the provided username and password against a stored hash in a file.

**Parameters:**

* username: A string representing the user's username.
* password: A string representing the user's password.

**Returns:**

* True: If authentication is successful.
* False: If authentication fails.

**Implementation:**

1. **Hashing Password:**
   * The provided password is hashed using the SHA-256 algorithm to ensure security.
   * The hashed password is stored as a hexadecimal string.
2. **Reading Authentication File:**
   * The function opens a file named AUTH\_FILE in read mode.
   * Each line in the file is assumed to contain a username and a hashed password, separated by a comma.
3. **Comparing Credentials:**
   * For each line in the file, the stored username and hashed password are extracted.
   * The provided username and hashed password are compared with the stored values.
4. **Authentication Success/Failure:**
   * If a match is found, a success message is printed, and True is returned.
   * If no match is found after processing all lines in the file, an error message is printed, and False is returned.

# Phase 4

## Integration of the Secure Communication Suite

**1. Client-Side Integration**

**1.1 User Authentication**

* **Purpose:** Ensures only authorized users can access the system.
* **Parameters:**
  + username: The username provided by the client.
  + password: The password provided by the client.
* **Functionality:**
  + Calls the authenticate\_user function to validate the user's credentials.
  + If authentication fails, the client exits; otherwise, the session proceeds.

**A computer screen with text on it

Description automatically generated**

1.2 **Message Encryption (AES)**

* **Purpose**: Encrypts the user-provided message using AES for secure transmission.
* **Parameters**:
  + message: The plaintext message provided by the user.
  + aes\_key: The AES key used for encryption.
  + iv: The initialization vector (IV) used for AES in CBC mode.
* **Functionality**:
  + Initializes an AESHandler object with the AES key and IV.
  + Calls aes\_handler.encrypt(message) to produce the encrypted message.

**A screenshot of a computer program

Description automatically generated**

1.3 **Key Exchange (RSA)**

* **Purpose**: Secures the AES key transmission using RSA encryption.
* **Parameters**:
  + aes\_key: The AES key to be encrypted.
  + username: The user's username (optional, encrypted for confidentiality).
  + server\_public\_key: The server's RSA public key.
* **Functionality**:
  + Encrypts the AES key and username with the server's public key using OAEP padding.
  + Prepares the encrypted AES key and username for transmission.

**A computer screen shot of a computer code

Description automatically generated**

1.4 **Data Packaging and Transmission**

* **Purpose**: Combines encrypted components for transmission over the network.
* **Parameters**:
  + encrypted\_aes\_key: The AES key encrypted with RSA.
  + encrypted\_message: The AES-encrypted message.
  + message\_hash: A SHA-256 hash of the plaintext message.
  + client\_socket: The TCP socket connection.
* **Functionality**:
  + Combines all components with a delimiter (b"||").
  + Send the packaged data to the server over the TCP socket.

**A screen shot of a computer program

Description automatically generated**

**2. Server-Side Integration**

2.1 **Function: create\_server\_keys()**

**Purpose**:

* Generate RSA keys for the server.
* Create and manage the server's certificate.

**Implementation**:

1. **RSA Key Generation**:
   * Checks if server\_private.pem and server\_public.pem files exist.
   * If not, generates a new RSA key pair (2048-bit) using rsa.generate\_private\_key().
   * Saves the private key in PKCS8 format to server\_private.pem.
   * Saves the public key in PEM format to server\_public.pem.
2. **Certificate Authority (CA) Creation**:
   * Checks if the CA certificate (ca\_cert.pem) and private key (ca\_key.pem) exist.
   * If not, calls create\_ca() to generate a self-signed CA certificate and saves it using save\_certificate().
3. **Server Certificate Generation**:
   * If the server certificate (server\_cert.pem) doesn’t exist:
     + Loads the CA certificate and private key.
     + Calls generate\_user\_certificate() to create and sign a certificate for the server.

Saves the server certificate to server\_cert.pem.

**Outputs**:

* Server RSA private key: server\_private.pem.
* Server RSA public key: server\_public.pem.
* CA certificate: ca\_cert.pem.
* Server certificate: server\_cert.pem

**A screen shot of a computer program

Description automatically generated**

**A computer screen with text and images

Description automatically generated**

**A screenshot of a computer code

Description automatically generated**

2.2 **Server Communication**

2.2.1 **Function: start\_server()**

**Purpose**:

* Set up the server to handle secure communication with clients.
* Receive, decrypt, and verify messages from clients.

**Implementation**:

**Certificate Validation**:

* Validates the server’s certificate (server\_cert.pem) against the CA certificate (ca\_cert.pem) using validate\_certificate().

A computer screen shot of a program code

Description automatically generated

**RSA Decryption Setup**:

* Loads the server's private RSA key (server\_private.pem).
* Initializes an RSAHandler instance with the private key for decryption.

A screen shot of a computer screen

Description automatically generated

**Socket Server Setup**:

* Sets up a TCP socket server to listen on localhost:12345.
* Accepts a client connection and establishes communication.

A computer screen shot of a program code

Description automatically generated

**Receiving and Processing Client Data**:

* Receives data from the client in a single transmission.
* Splits the received data into:
  + encrypted\_aes\_key: AES key encrypted with the server's RSA public key.
  + encrypted\_username: Username encrypted with the server's RSA public key.
  + encrypted\_message: AES-encrypted message.
  + received\_hash: SHA-256 hash of the plaintext message.
* Logs data lengths and formats for debugging.

A screenshot of a computer program

Description automatically generated

**Decrypting Components**:

* Decrypts the AES key using RSA private key.
* Decrypts the username using RSA private key.
* Uses the decrypted AES key and a predefined IV (b"RandomIV12345678") to decrypt the message with AESHandler.

A computer screen shot of a program

Description automatically generated

**Message Integrity Verification**:

* Computes the SHA-256 hash of the decrypted message.
* Compares the computed hash with the received\_hash to verify integrity.
* Logs the integrity status (Intact or Compromised).

A screen shot of a computer

Description automatically generated

**2.2.2 Workflow of the server**

1. **Receive** **Data**:

* The server receives a single payload containing:
  + encrypted\_aes\_key
  + encrypted\_username
  + encrypted\_message
  + received\_hash
* Data is separated using a delimiter (b"||").

2. **Decryption**:

* AES **Key**:
  + Decrypts using RSA private key.
* **Username**:
  + Decrypts using RSA private key to retrieve the plaintext username.
* **Message**:
  + Decrypts the AES-encrypted message using AESHandler.

**3. Verification**:

* Computes the hash of the decrypted message using SHA-256.
* Compares the computed hash with the received\_hash.

**3. Security Features**

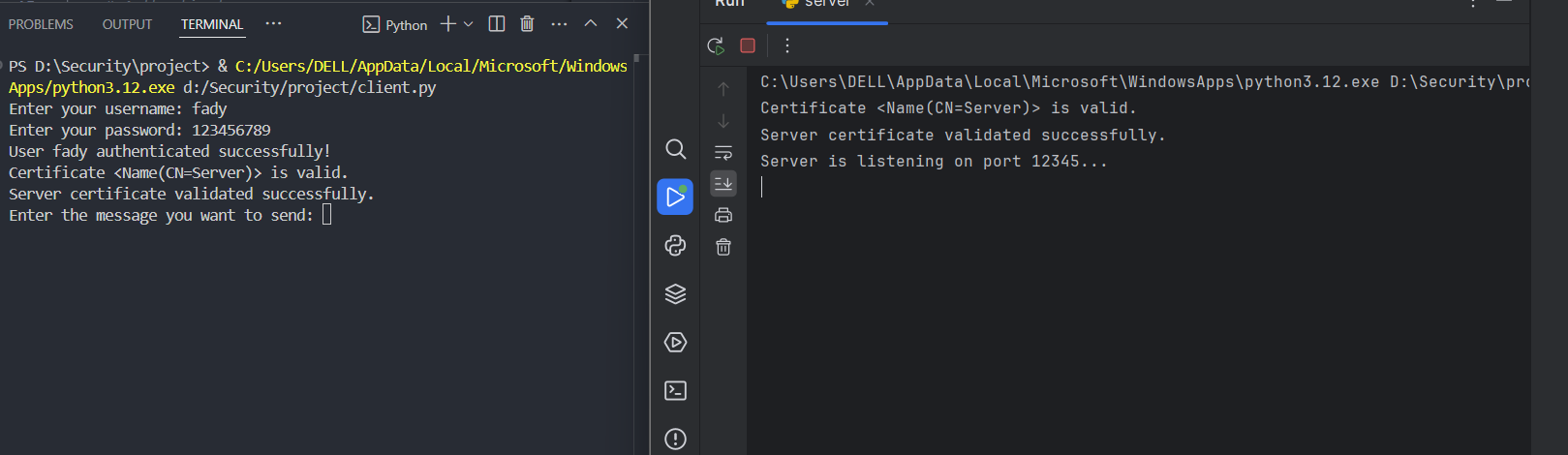
1. **Authentication**:
   * Validates its certificate using the CA’s certificate to establish trust.
   * Decrypts the username to verify the client’s identity.
2. **Confidentiality**:
   * Decrypts the AES key to ensure secure message decryption.
   * Messages remain confidential during transmission and storage.
3. **Integrity**:
   * Verifies the SHA-256 hash to ensure messages haven’t been altered.
4. **Key Management**:
   * Uses RSA for secure AES key exchange.

**4. Testing**

**Test case#1:**

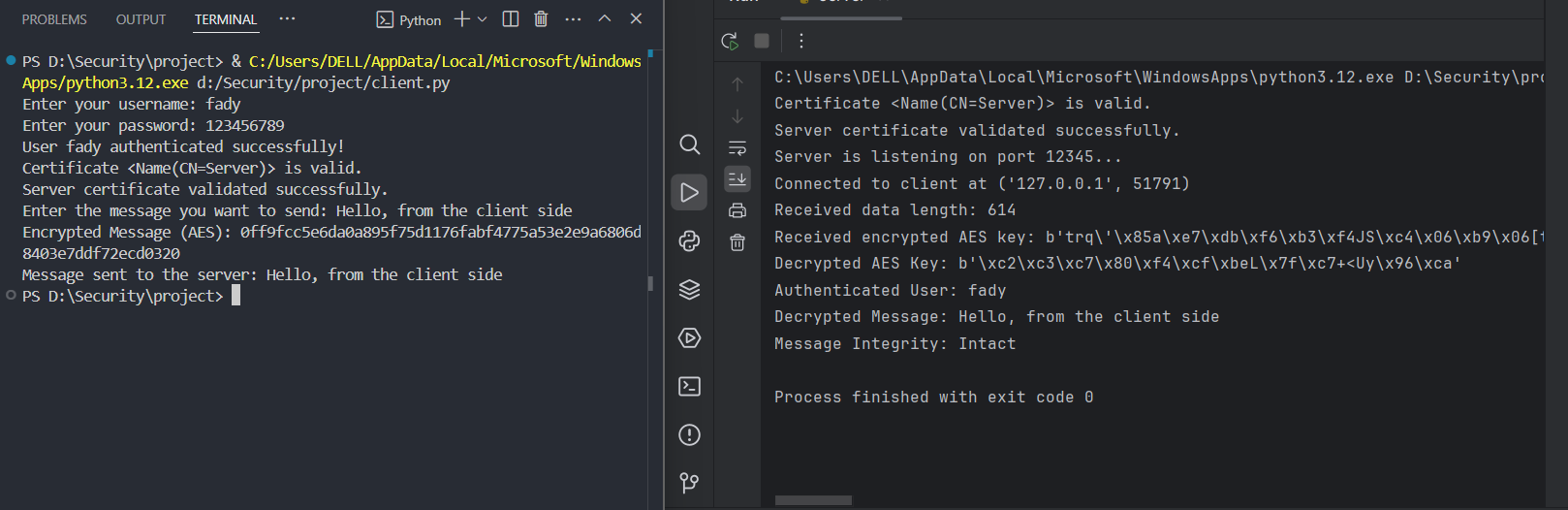
**Valid User Login**

* **Purpose**: To validate that a user can log in successfully with correct credentials.
* **Steps**:
  1. Run the client.py script.
  2. Enter the username fady.
  3. Enter the password 123456789.
* **Expected Result:**
  1. The client displays: User fady authenticated successfully!
  2. The client proceeds to the message input step.
* **Result:**
  1. As seen in the screenshot, the user was successfully authenticated.
  2. The client proceeded to the message input step.



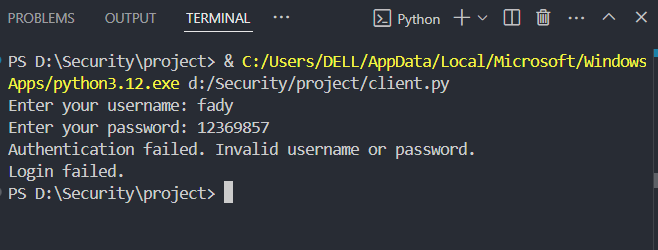
Server Certificate Validation and Message Encryption/Decryption

* Purpose: To validate that the client successfully validates the server's certificate and ensures secure communication through AES encryption and decryption.
* Steps:
  1. Ensure that the server's certificate is validated by the client.
  2. Input a test message: Hello, from the client side.
  3. Observe the server's decryption process and message integrity verification.
* Expected Result:
  1. The client validates the server's certificate and displays: Server certificate validated successfully.
  2. The client encrypts the message using AES and sends it to the server.
  3. The server decrypts the message and displays: Decrypted Message: Hello, from the client side.
  4. The server verifies the message integrity and displays: Message Integrity: Intact.
* Actual Result:
  1. As seen in the screenshot, the client successfully validated the server's certificate.
  2. The message was encrypted, transmitted to the server, and decrypted successfully.
  3. The server verified the integrity of the message, confirming it was intact.



**Test Case #2: Authentication with Wrong Password**

* **Purpose**: To validate that the system denies access when an incorrect password is provided.
* **Steps**:
  1. Run the client.py script.
  2. Enter the username: fady.
  3. Enter an incorrect password: 12369857.
* **Expected Result:**
  1. The client displays: Authentication failed. Invalid username or password.
  2. The client terminates the login process and does not proceed further.
* **Actual Result:**
  1. As seen in the screenshot, the client displayed: Authentication failed. Invalid username or password.
  2. The login process was terminated, and the user was unable to proceed.
* **Conclusion**: Test passed as the expected behavior matches the actual result.



**Test Case #3: User Registration and Certificate Generation**

**Purpose**: To validate that a new user can register successfully and a valid certificate is generated for the user.

**Steps**:

1. Run the main.py script.
2. Choose option 1 (Register User) from the menu.
3. Enter a username: maged.
4. Enter a password: 147258369.
5. Check if the certificate is saved at the specified location (crypto/certificates/maged.crt).
6. Open the generated certificate to verify its details.

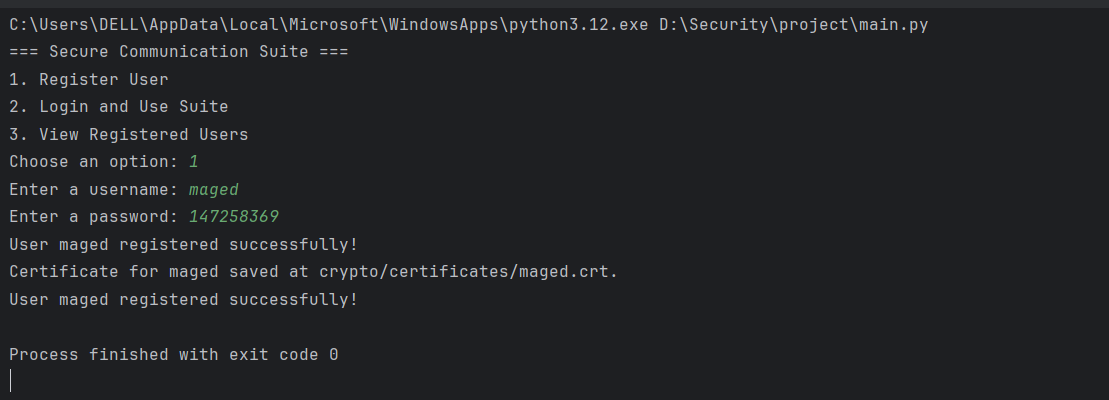
**Expected Result:**

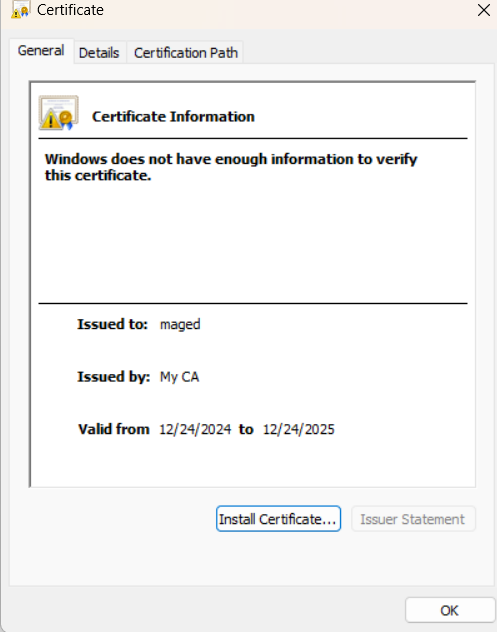
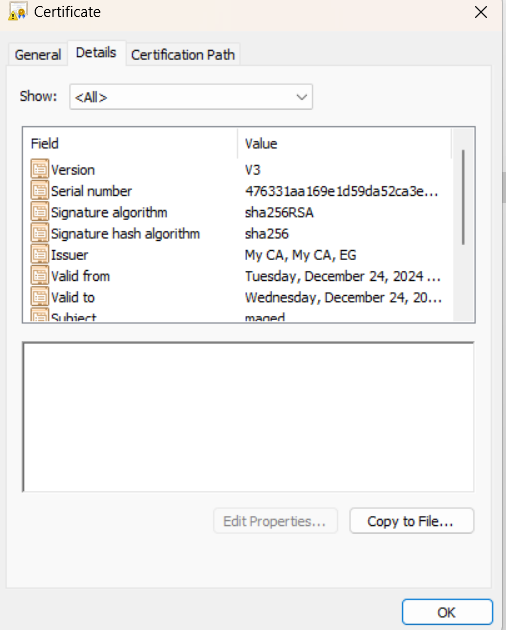
* The client displays: User maged registered successfully!
* A certificate for the user maged is generated and saved.
* The certificate details include:
  + Issued to: maged
  + Issued by: My CA
  + Validity Period: 12/24/2024 to 12/24/2025
  + Signature Algorithm: sha256RSA

**Actual Result:**

* As seen in the screenshots, the user maged was registered successfully.
* A certificate for maged was generated and saved.
* The certificate details matched the expected values.

**Conclusion**: Test passed as the expected behavior matches the actual result.





## Real Tools or Services similar to our suite:

**Cryptography Tools and Libraries**

**1**. **OpenSSL**: Widely used for secure communication, certificate generation, and encryption.

* Cost: Free (Open Source).
* Use: Replace custom certificate generation and validation with OpenSSL for standardized security.

A screenshot of a application delivery

Description automatically generated

**2. AWS KMS(Key Management Service)**

Cost: Each AWS KMS key that you create in AWS KMS costs $1/month (prorated hourly). The $1/month charge is the same for symmetric keys, asymmetric keys, HMAC keys, multi-Region keys (each primary and each replica multi-Region key), keys with imported key material, AWS KMS provides a free tier of 20,000 requests/month calculated.

Use: Secure key generation, management, and storage in a cloud environment.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Cost Analysis for a prototype

1**. Development Costs**

Adjusted Developer Time

* Estimation:
  + Number of developers: 2
  + Reduced hours: 40 hours per developer.
  + Hourly rate: $25/hour.
* Cost Calculation:
  + 2 developers × 40 hours × $25/hour = $2,000

2. **Hosting and Infrastructure**

**Cloud Hosting (Server Deployment)**

* **AWS EC2 t2.micro (Linux Instance):**
  + Monthly cost: ~$9.62 (on-demand instance in free-tier region).

**DNS and Domain Name**

* Average domain name cost: $12–$15/year for .com domains.

**Additional Costs (Storage and Bandwidth)**

* **AWS S3 (File Storage for Certificates and Logs):**
  + $0.023/GB per month.
  + Estimated usage: 1 GB/month = ~$0.03.
* **Bandwidth Costs:**
  + Example: AWS Data Transfer Out (first 1 GB free, then ~$0.09/GB).
  + Estimated monthly usage: 10 GB = ~$0.90.

**Total Hosting Costs:**

* Minimum: ~$6–$10/month for a small-scale deployment.

**3. Cryptographic Tools and Services**

**Key Management**

* AWS KMS (Key Management Service):
  + Cost: $1 per 10,000 key operations.
  + Prototype Usage: ~1,000 operations/month = ~$0.10.

**Certificate Management**

* Let’s Encrypt (Free):
  + Issue SSL/TLS certificates at no cost.

**Secure Storage**

* **HashiCorp Vault (Open Source):**
  + Free for open-source usage.

**Total Tool Costs:**

* Minimal (~$0.10/month).

**4. Database for User Management**

**Firebase Authentication**

* **Free Tier:**
  + Up to 50,000 users per month included for free.

**Amazon RDS (Relational Database Service)**

* Monthly cost for small instances:
  + ~$12–$15/month for PostgreSQL or MySQL.

Total Database Costs:

* ~$0–$15/month, depending on number of users.

**5. Development Tools and Licenses**

**Libraries and Frameworks**

* Python Cryptography Library: Free (Open Source).
* Additional Dependencies: Free.

**Development Environment**

* GitHub (Free tier for personal use or $4/month for Pro).

**Total Development Tools Cost:**

* ~$0–$4/month.

**Initial Development Costs (One-Time)**

The total cost for development involves developer hours and domain registration. With 2 developers working 40 hours each at $25/hour, the development cost is $2,000. Additionally, domain registration costs range from $12 to $15 per year. Therefore, the total one-time cost for development is approximately **$2,012 to $2,015**.

**Monthly Operational Costs (Prototype)**

For ongoing operations, the estimated monthly costs are:

* **Cloud Hosting:** $6 to $10 per month for a basic hosting solution.
* **Bandwidth and Storage:** Approximately $1 per month for small-scale data transfer and storage.
* **Key Management:** Around $0.10 per month for basic usage of services like AWS KMS.
* **Database:** Costs range from $0 (using Firebase free tier) to $15 per month for a small Amazon RDS instance.
* **Development Tools:** Free for basic needs or $4 per month for Pro tools like GitHub.

In total, the monthly operational cost for the prototype is approximately **$7 to $30 per month**, depending on the chosen configurations, services and number of users.

## Business Value to the Market

**Market Potential**

* **Target Audience:**
  + Small businesses, educational institutions, and startups.
* **Use Cases:**
  + Securing communications, user authentication, encrypted file sharing.
* **Pricing Model:**
  + Charge small businesses $50–$100/month for a fully managed solution.

**Profitability**

* For a managed service:
  + If targeting 100 clients at $50/month: $5,000/month revenue.
  + Estimated operational cost for managing 100 clients: ~$500/month.
  + **Profit Margin:** ~$4,500/month.