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** |

# 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 (e.g., CBC, CTR, GCM).
* **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.
* **Secure Communication Protocols:** Use TLS/SSL for secure communication.
* **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.
* **Deployment:** Deploy the application to a secure server environment.

## Software Requirements

### Functional Requirements

1. **Encryption/Decryption:**
   * Implement symmetric encryption algorithms (e.g., AES) for confidentiality.
   * Implement asymmetric encryption algorithms (e.g., RSA, ECC) 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 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.
   * Implement automated testing and continuous integration/continuous delivery (CI/CD) practices.

## 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