



**AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
CREDIT HOURS ENG.
PROGRAM**

**Computer engineering and
software systems**



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
Overview	3
Project Plan.....	3
Software Requirements	4
Functional Requirements	4
Non-Functional Requirements	5
Design Documents	6
Use case Diagram.....	6
Phase 2	7
Development of Cryptographic Modules	7
Block Cipher	7
Public key cryptosystem	8
Hashing.....	13
Phase 3	15
Key management Moule	15
Authentication Module	18
Phase 4	19
Integration of the Secure Communication Suite.....	19
Real Tools or Services similar to our suite:	33
Cost Analysis for a prototype	35
Business Value to the Market	38

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.

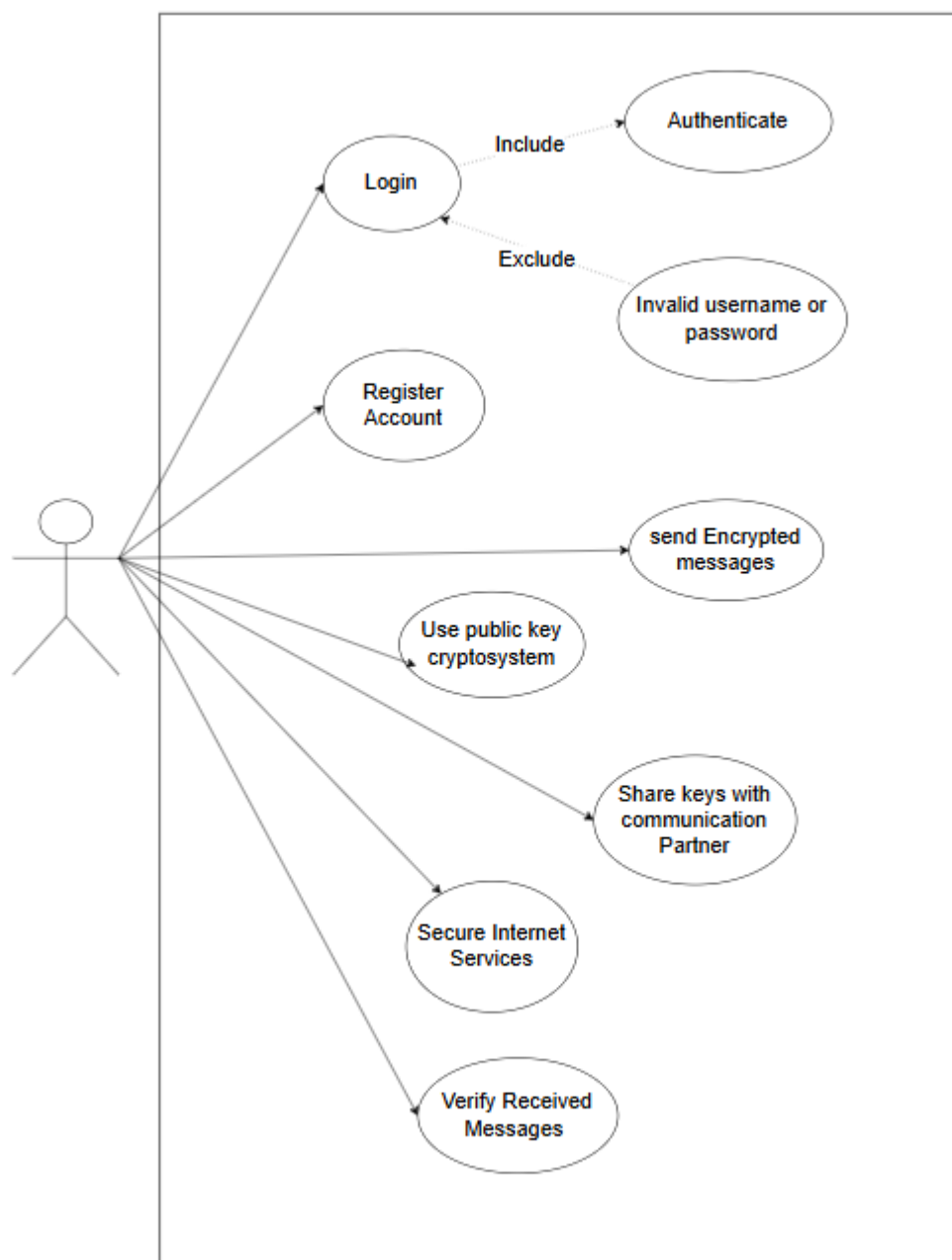
- Consider distributed architectures for scalability.

6. Maintainability:

- Write clean, well-structured, and well-documented code.
- Use modular design principles.

Design Documents

Use case Diagram



Phase 2

Development of Cryptographic Modules

Block Cipher

As for the Block Cipher we used AES

```
✓ from Crypto.Cipher import AES
  from Crypto.Util.Padding import pad, unpad

✓ class AESHandler:
  ✓ def __init__(self, key, iv):
    self.key = key
    self.iv = iv
    self.cipher = AES.new(self.key, AES.MODE_CBC, self.iv)

  ✓ def encrypt(self, plain_text):
    """Encrypts plaintext using AES in CBC mode."""
    plain_text_padded = pad(plain_text.encode(), AES.block_size)
    cipher_text = self.cipher.encrypt(plain_text_padded)
    return cipher_text

  ✓ def decrypt(self, cipher_text):
    """Decrypts ciphertext using AES in CBC mode."""
    plain_text_padded = self.cipher.decrypt(cipher_text)
    plain_text = unpad(plain_text_padded, AES.block_size).decode()
    return plain_text
```

```
def generate_iv(block_size=16):
    """Generate a random initialization vector (IV)."""
    return token_bytes(block_size)
```

```
def generate_aes_key(key_size=16):
    💡 """Generate a random AES key."""
    return get_random_bytes(key_size)
```

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

```
def generate_rsa_keys():  
    """Generate RSA public and private keys using cryptography."""  
    private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)  
    public_key = private_key.public_key()  
    return public_key, private_key
```

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.

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
```

```
class RSAHandler:
```

```
    def __init__(self, public_key=None, private_key=None):
        self.public_key = public_key
        self.private_key = private_key
```

```
    def encrypt(self, data):
        """Encrypt data with RSA public key."""
        ciphertext = self.public_key.encrypt(
            data.encode(), # Convert to bytes
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return ciphertext
```

```
    def decrypt(self, data):
        """Decrypt data with RSA private key."""
        try:
            plaintext = self.private_key.decrypt(
                data,
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None
                )
            )
            return plaintext
        except Exception as e:
            print(f"Decryption failed: {e}")
            return None
```

```
def sign(self, message):  
    """Sign a message using the private key."""  
    signature = self.private_key.sign(  
        message.encode(), # Convert to bytes  
        padding.PKCS1v15(),  
        hashes.SHA256()  
    )  
    return signature
```

```
def verify(self, message, signature):  
    """Verify the signature of a message using the public key."""  
    try:  
        self.public_key.verify(  
            signature,  
            message.encode(), # Convert to bytes  
            padding.PKCS1v15(),  
            hashes.SHA256()  
        )  
        return True  
    except Exception as e:  
        return False
```

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

```
def compute_sha256(data):
    """Generate SHA-256 hash of the given data."""
    sha256_hash = hashlib.sha256()
    sha256_hash.update(data.encode()) # Convert data to bytes before hashing
    return sha256_hash.hexdigest()
```

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.

```
def compute_md5(data):
    """Generate MD5 hash of the given data."""
    md5_hash = hashlib.md5()
    md5_hash.update(data.encode()) # Convert data to bytes before hashing
    return md5_hash.hexdigest()
```

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.

```
def verify_hash(data, given_hash, algorithm="sha256"):
    """Verify if the hash of the data matches the given hash."""
    if algorithm == "sha256":
        return compute_sha256(data) == given_hash
    elif algorithm == "md5":
        return compute_md5(data) == given_hash
    else:
        raise ValueError("Unsupported hashing algorithm. Use 'sha256' or 'md5'.")
```

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

```
from Crypto.Random import get_random_bytes
from secrets import token_bytes
import os
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

def generate_aes_key(key_size=16):
    """Generate a random AES key."""
    return get_random_bytes(key_size)

def generate_iv(block_size=16):
    """Generate a random initialization vector (IV)."""
    return token_bytes(block_size)

def generate_rsa_keys():
    """Generate RSA public and private keys using cryptography."""
    private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
    public_key = private_key.public_key()
    return public_key, private_key

def generate_and_store_rsa_keys(username):
    """Generate RSA keys and store them in the user's directory."""
    private_key_file = f"data/keys/{username}_private.pem"
    public_key_file = f"data/keys/{username}_public.pem"

    if not (os.path.isfile(private_key_file) and os.path.isfile(public_key_file)):
        public_key, private_key = rsa.newkeys(1024)
        with open(private_key_file, "wb") as f:
            f.write(private_key.save_pkcs1("PEM"))
        with open(public_key_file, "wb") as f:
            f.write(public_key.save_pkcs1("PEM"))
        print(f"RSA keys generated for {username}.")
    else:
        print(f"RSA keys for {username} already exist.")
```

```
def load_rsa_keys(username):
    """Load RSA keys for the given user."""
    private_key_file = f"data/keys/{username}_private.pem"
    public_key_file = f"data/keys/{username}_public.pem"

    with open(private_key_file, "rb") as f:
        private_key = rsa.PrivateKey.load_pkcs1(f.read())
    with open(public_key_file, "rb") as f:
        public_key = rsa.PublicKey.load_pkcs1(f.read())
    return public_key, private_key
```

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

```
def authenticate_user(username, password):  
    """Authenticate user by validating username and hashed password."""  
    hashed_password = hashlib.sha256(password.encode()).hexdigest()  
    with open(AUTH_FILE, "r") as f:  
        for line in f:  
            stored_username, stored_hash = line.strip().split(",")  
            if stored_username == username and stored_hash == hashed_password:  
                print(f"User {username} authenticated successfully!")  
                return True  
    print("Authentication failed. Invalid username or password.")  
    return False
```

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.

```

1 usage  YousefFayez20
def start_client():
    # Authenticate user
    username = input("Enter your username: ").strip()
    password = input("Enter your password: ").strip()

    if not authenticate_user(username, password):
        print("Login failed.")
        return

```

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.

```
# Generate AES key
aes_key = generate_aes_key()

# Prompt the user for the message they want to send
message = input("Enter the message you want to send: ").strip()

# Predefined IV for AES encryption
iv = b"RandomIV12345678"
aes_handler = AESHandler(aes_key, iv)

# Encrypt the message using AES
encrypted_message = aes_handler.encrypt(message)
print(f"Encrypted Message (AES): {encrypted_message.hex()}")
```

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.

```

# Encrypt AES key with the server's RSA public key
encrypted_aes_key = public_key.encrypt(
    aes_key,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Encrypt the username for confidentiality
encrypted_username = public_key.encrypt(
    username.encode(),
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

```

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.

```

69     # Compute hash of the message for integrity verification
70     message_hash = compute_sha256(message)
71
72     # Send the data to the server
73     client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
74     client_socket.connect(("localhost", 12345))
75
76     # Send encrypted AES key, encrypted username, encrypted message, and the message hash
77     client_socket.send(
78         encrypted_aes_key + b"||" + encrypted_username + b"||" + encrypted_message + b"||" + message_hash.encode())
79
80     print(f"Message sent to the server: {message}")
81     client_socket.close()
82
83
84 # Client-side entry point
85 if __name__ == "__main__":
86     start_client()

```

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

○

```
def create_server_keys():
    """Generate server's RSA key pair, save the public key, and create a server certificate."""
    private_key_file = "server_private.pem"
    public_key_file = "server_public.pem"
    server_cert_path = "crypto/certificates/server_cert.pem"
    ca_cert_path = "crypto/certificates/ca_cert.pem"
    ca_key_path = "crypto/certificates/ca_key.pem"

    # Generate server RSA keys if not already created
    if not os.path.exists(private_key_file) or not os.path.exists(public_key_file):
        private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
        public_key = private_key.public_key()

        with open(private_key_file, "wb") as priv_file:
            priv_file.write(
                private_key.private_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PrivateFormat.PKCS8,
                    encryption_algorithm=serialization.NoEncryption(),
                )
            )
```

```

        with open(public_key_file, "wb") as pub_file:
            pub_file.write(
                public_key.public_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PublicFormat.SubjectPublicKeyInfo,
                )
            )
        print("Server RSA keys generated and saved.")

# Check if the CA exists; create if not
if not os.path.exists(ca_cert_path) or not os.path.exists(ca_key_path):
    ca_key, ca_cert = create_ca()
    save_certificate(ca_cert, ca_cert_path)
    with open(ca_key_path, "wb") as ca_key_file:
        ca_key_file.write(
            ca_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption(),
            )
        )
    print("CA created and saved.")

```

```

# Load CA certificate and key
with open(ca_cert_path, "rb") as f:
    ca_cert = x509.load_pem_x509_certificate(f.read())

with open(ca_key_path, "rb") as f:
    ca_key = serialization.load_pem_private_key(f.read(), password=None)

# Generate server certificate if not already created
if not os.path.exists(server_cert_path):
    with open(private_key_file, "rb") as priv_file:
        server_private_key = serialization.load_pem_private_key(priv_file.read(), password=None)

    generate_user_certificate(username="Server", server_private_key, ca_key, ca_cert, cert_filename=server_cert_path)
    print(f"Server certificate generated and saved to {server_cert_path}.")

```

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().

```

def start_server():
    ca_cert_path = "crypto/certificates/ca_cert.pem"
    server_cert_path = "crypto/certificates/server_cert.pem"

    if not validate_certificate(server_cert_path, ca_cert_path):
        print("Server certificate validation failed. Exiting...")
        return

    print("Server certificate validated successfully.")

```

RSA Decryption Setup:

- Loads the server's private RSA key (server_private.pem).
- Initializes an RSAHandler instance with the private key for decryption.

```

# Load server's private key for RSA decryption
with open("server_private.pem", "rb") as f:
    private_key = serialization.load_pem_private_key(f.read(), password=None)

# RSA handler with the private key for decryption
rsa_handler = RSAHandler(public_key=None, private_key=private_key)

```

Socket Server Setup:

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

```

# Server socket setup
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(("localhost", 12345))
server_socket.listen(1)
print("Server is listening on port 12345...")

conn, addr = server_socket.accept()
print(f"Connected to client at {addr}")

```

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.

```

try:
    # Receive data from the client
    data = conn.recv(4096)
    if not data:
        print("No data received from the client.")
        return

    # Split the data: encrypted AES key, encrypted username, encrypted message, and message hash
    # Debugging: Print received data lengths
    print(f"Received data length: {len(data)}")

    # Properly handle splitting using a more robust protocol (e.g., length-prefixed data)
    try:
        encrypted_aes_key, encrypted_username, encrypted_message, received_hash = data.split(b"||")
    except ValueError as e:
        print(f"Data splitting error: {e}")
        return

    print(f"Received encrypted AES key: {encrypted_aes_key}")

```

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.

```

# Decrypt the AES key using RSA private key
aes_key = rsa_handler.decrypt(encrypted_aes_key)
if aes_key is None:
    print("Failed to decrypt the AES key.")
    return
print(f"Decrypted AES Key: {aes_key}")

# Decrypt the username
username = rsa_handler.decrypt(encrypted_username).decode() # Decrypt the username
print(f"Authenticated User: {username}")

# Now handle message decryption using AES
iv = b"RandomIV12345678" # Predefined IV
aes_handler = AESHandler(aes_key, iv)

try:
    # Decrypt the message using AES
    decrypted_message = aes_handler.decrypt(encrypted_message)
    print(f"Decrypted Message: {decrypted_message}")
except ValueError as e:
    print(f"Error during decryption: {e}")

```

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).

```
# Verify message integrity using SHA-256
computed_hash = compute_sha256(decrypted_message)
integrity_check = computed_hash == received_hash.decode()
print(f"Message Integrity: {'Intact' if integrity_check else 'Compromised'}")
```

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:

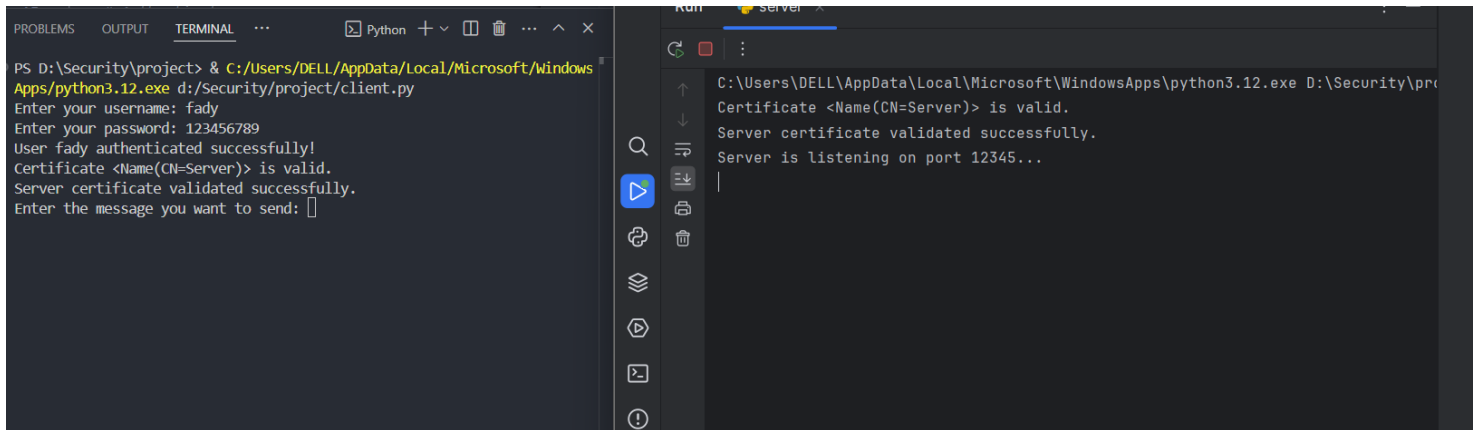
- Uses RSA for secure AES key exchange.

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:**
 - The client displays: User fady authenticated successfully!
 - The client proceeds to the message input step.
- **Result:**
 - As seen in the screenshot, the user was successfully authenticated.
 - The client proceeded to the message input step.



The screenshot shows a VS Code interface with a terminal and an output window. The terminal window on the left displays the following commands and output:

```
PS D:\Security\project> & C:/Users/DELL/AppData/Local/Microsoft/Windows  
Apps/python3.12.exe d:/Security/project/client.py  
Enter your username: fady  
Enter your password: 123456789  
User fady authenticated successfully!  
Certificate <Name(CN=Server)> is valid.  
Server certificate validated successfully.  
Enter the message you want to send: |
```

The output window on the right displays the following messages:

```
C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.12.exe D:\Security\pro  
Certificate <Name(CN=Server)> is valid.  
Server certificate validated successfully.  
Server is listening on port 12345...
```

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:
 - The client validates the server's certificate and displays: Server certificate validated successfully.
 - The client encrypts the message using AES and sends it to the server.
 - The server decrypts the message and displays: Decrypted Message: Hello, from the client side.
 - The server verifies the message integrity and displays: Message Integrity: Intact.
- Actual Result:
 - As seen in the screenshot, the client successfully validated the server's certificate.

- The message was encrypted, transmitted to the server, and decrypted successfully.
- The server verified the integrity of the message, confirming it was intact.

The screenshot shows a VS Code interface with a terminal and an output window. The terminal on the left shows the execution of a Python script from the command prompt. The output window on the right shows the server's logs.

```

PS D:\Security\project> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.12.exe d:/Security/project/client.py
Enter your username: fady
Enter your password: 123456789
User fady authenticated successfully!
Certificate <Name(CN=Server)> is valid.
Server certificate validated successfully.
Enter the message you want to send: Hello, from the client side
Encrypted Message (AES): 0ff9fcc5e6da0a895f75d1176fabf4775a53e2e9a6806d8403e7ddf72ecd0320
Message sent to the server: Hello, from the client side
PS D:\Security\project>

C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.12.exe D:\Security\project\server.py
Certificate <Name(CN=Server)> is valid.
Server certificate validated successfully.
Server is listening on port 12345...
Connected to client at ('127.0.0.1', 51791)
Received data length: 614
Received encrypted AES key: b'trq'\x85a\xe7\xdb\xf6\xb3\xf4JS\xc4\x06\xb9\x06[
Decrypted AES Key: b'\xc2\xc3\xc7\x80\xf4\xcf\xbeL\x7f\xc7+<Uy\x96\xca'
Authenticated User: fady
Decrypted Message: Hello, from the client side
Message Integrity: Intact

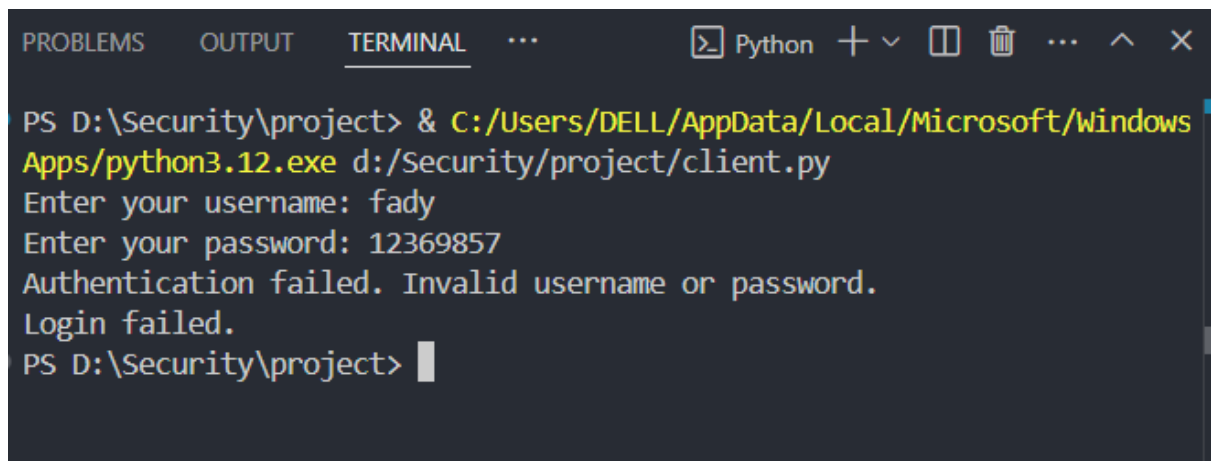
Process finished with exit code 0

```

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:**
 - The client displays: Authentication failed. Invalid username or password.
 - The client terminates the login process and does not proceed further.
- **Actual Result:**

- As seen in the screenshot, the client displayed: Authentication failed. Invalid username or password.
- The login process was terminated, and the user was unable to proceed.
- **Conclusion:** Test passed as the expected behavior matches the actual result.



The screenshot shows a Windows terminal window with the title bar 'Python' and standard window controls. The terminal content is as follows:

```
PS D:\Security\project> & C:/Users/DELL/AppData/Local/Microsoft/Windows  
Apps/python3.12.exe d:/Security/project/client.py  
Enter your username: fady  
Enter your password: 12369857  
Authentication failed. Invalid username or password.  
Login failed.  
PS D:\Security\project> █
```

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

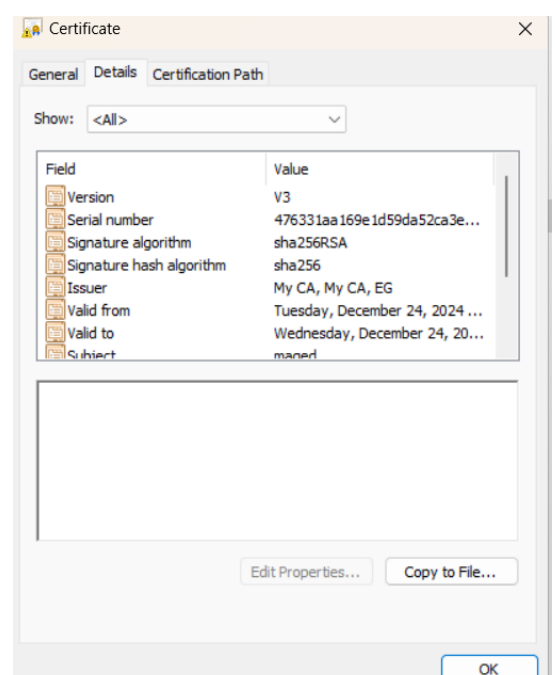
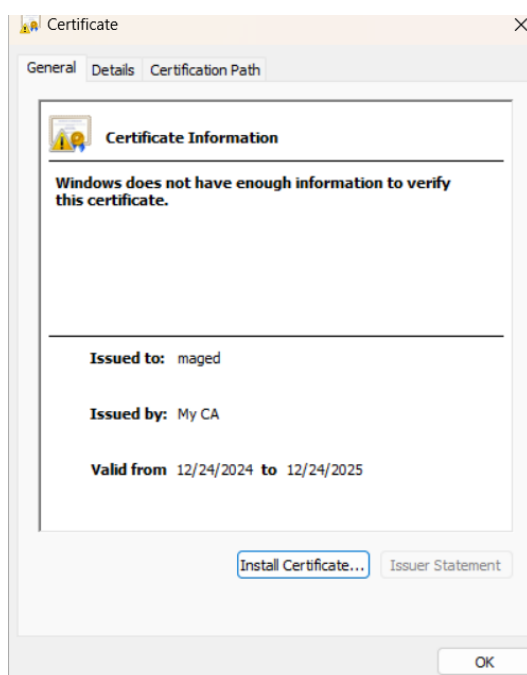
Actual Result:

- 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.

```
C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.12.exe D:\Security\project\main.py
=== Secure Communication Suite ===
1. Register User
2. Login and Use Suite
3. View Registered Users
Choose an option: 1
Enter a username: maged
Enter a password: 147258369
User maged registered successfully!
Certificate for maged saved at crypto/certificates/maged.crt.
User maged registered successfully!

Process finished with exit code 0
```



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.

Application Delivery

OpenSSL can play a crucial role in a secure application delivery strategy, particularly when it comes to ensuring the confidentiality, integrity, and authenticity of data transmitted over networks.

It is commonly used for these use cases:

- Secure communication between clients and servers
- Data encryption
- Certificate management
- Data integrity and authentication
- Random number generation
- Secure protocols and algorithms
- Secure web server deployment

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.

Get started with AWS KMS

Benefits

Manage keys and define policies from a single point

Centrally manage keys and define policies across integrated services and applications from a single point.

Encrypt data within your applications

Encrypt data within your applications with the AWS Encryption SDK data encryption library.

Perform signing operations

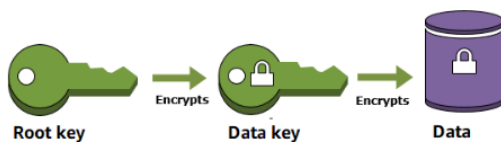
Perform signing operations using asymmetric key pairs to validate digital signatures.

Securely generate HMACs

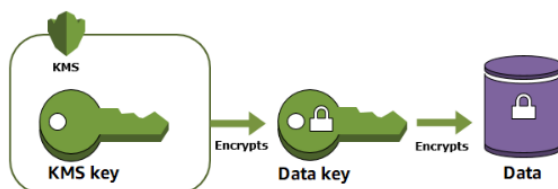
Securely generate hash-based message authentication codes (HMACs) that ensure message integrity and authenticity.

Why use AWS KMS?

When you encrypt data, you need to protect your encryption key. If you encrypt your key, you need to protect its encryption key. Eventually, you must protect the highest level encryption key (known as a *root key*) in the hierarchy that protects your data. That's where AWS KMS comes in.



AWS KMS protects your root keys. KMS keys are created, managed, used, and deleted entirely within AWS KMS. They never leave the service unencrypted. To use or manage your KMS keys, you call AWS KMS.



Additionally, you can create and manage [key policies](#) in AWS KMS, ensuring that only trusted users have access to KMS keys.

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:
 - $2 \text{ developers} \times 40 \text{ hours} \times \$25/\text{hour} = \$2,000$

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.