Project Phase 4

CSE451, Computer and Network Security

Date: 15/ 5 / 2024

Group 17

Aya Ahmed Gamal 19P1689

Karen Alber Farid 19P8948

Shaimaa Mohamed 19P7484

Contents

1.0 Introduction:

Ensuring the security and confidentiality of sensitive information is crucial in the current digital era. There has never been a more pressing need for secure communication channels because cyber threats are always changing and getting more complex. The Secure Communication Suite is being developed to address these issues. This suite is an all-inclusive application that combines different security protocols and cryptographic approaches to offer a reliable and safe communication environment.

Many elements in the Secure Communication Suite are intended to safeguard data both while it is in transit and while it is at rest. In order to ensure that data is encrypted using a shared secret key that only authorized parties hold, it integrates block ciphers for symmetric encryption. Asymmetric encryption also uses public-key cryptosystems, which enable safe key exchange between parties that never exchanged keys before.

Another essential element of the Secure Communication Suite is authentication techniques, which enable user identity verification. This guarantees that only authorized users may access the suite's capabilities and helps prevent unauthorized access to sensitive data.

All in all, the Secure Communication Suite offers a complete solution for safeguarding Internet services, including a variety of security protocols and cryptographic methods to preserve data and guarantee the authenticity, integrity, and confidentiality of communications.

2.0 Key Features:

1. Block Cypher Module: With the use of symmetric encryption methods like DES (Data Encryption Standard) and AES (Advanced Encryption Standard), this module can both encrypt and decode data. These methods guarantee that data is encrypted and only the sender and the recipient know the secret key.
2. The Public Key Cryptosystem Module is responsible for implementing asymmetric encryption techniques, including Elliptic Curve Cryptography (ECC) and Rivest-Shamir-Adleman (RSA). With asymmetric encryption, secrecy and authenticity can be guaranteed even when communicating securely between parties that haven't shared keys before.
3. Hashing Module: The hashing module creates unique checksums for data using techniques like MD5 or SHA-256. In order to confirm that the data has not been altered during transmission or storage, these checksums are utilized.
4. The Key Management Module: It offers safe techniques for creating, distributing, and storing keys. It offers procedures for safely exchanging keys between parties and guarantees that keys are shielded from unwanted access.
5. Module for Authentication: This module applies procedures for authentication which is password. This guards against unauthorized access by guaranteeing that only authorized users can access the secure communication suite.
6. Internet Services Security Module: To secure data for Internet services, this module uses cryptography modules. It guarantees the confidentiality and integrity of data communicated over the internet by guarding against eavesdropping and tampering.

3.0 Implementation:

3.1 Block Cipher Module:

```
# AES Encryption and Decryption with EAX mode
def encrypt_with_aes(message, key):
    cipher = AES.new(key, AES.MODE_EAX)
    ciphertext, tag = cipher.encrypt_and_digest(message)
    return cipher.nonce + tag + ciphertext

def decrypt_with_aes(ciphertext, key):
    nonce = ciphertext[:16]
    tag = ciphertext[16:32]
    actual_ciphertext = ciphertext[32:]
    cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    plaintext = cipher.decrypt_and_verify(actual_ciphertext, tag)
    return plaintext
```

Encryption with AES and EAX Mode

In the **encrypt_with_aes** function, a new AES cipher object is created with the provided key and EAX mode. EAX mode ensures both encryption (confidentiality) and authentication (integrity). The plaintext message is then encrypted and a message authentication code (MAC) tag is generated using the **encrypt_and_digest** method. The function concatenates the nonce (a random value used only once for this encryption session), the tag, and the ciphertext, and returns this combined byte array. The nonce is critical for decryption as it ensures the uniqueness of the encryption operation, preventing replay attacks.

Decryption with AES and EAX Mode

The **decrypt_with_aes** function begins by extracting the nonce, tag, and actual ciphertext from the combined byte array. It first retrieves the nonce from the first 16 bytes, followed by the tag from the next 16 bytes, and the remainder is the actual encrypted message. A new AES cipher object is then created with the provided key and EAX mode, using the extracted nonce. The function then decrypts the actual ciphertext and verifies its integrity using the **decrypt_and_verify** method, which checks the authenticity of the message using the tag. If the tag does not match, indicating potential tampering, an exception is raised. If the tag matches, the original plaintext message is returned.

3.2 Public Key Cryptosystem Module:

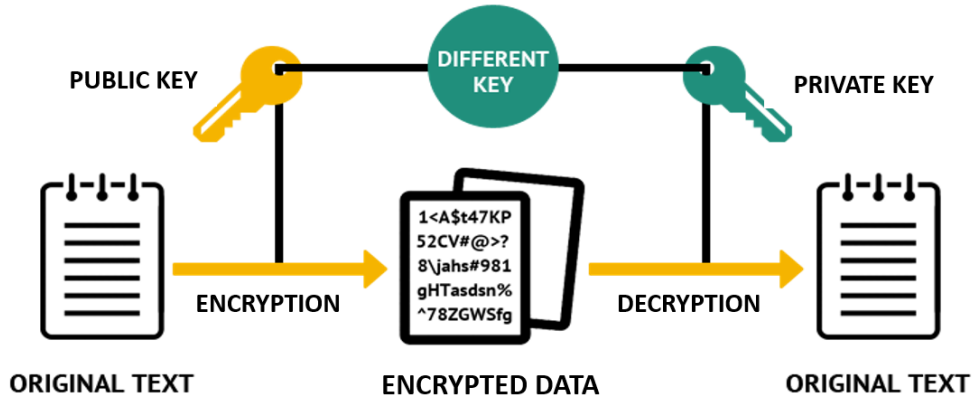
RSA Technique:

RSA allows you to secure messages before you send them. RSA signing and verification are cryptographic processes used to ensure the authenticity, integrity, and non-repudiation of data. RSA encryption relies on few basic assets and quite a bit of math. These elements are required:

- A public key (e)
- A private key (d)
- Two prime numbers (P and Q), multiplied (N)

Security relies on the assumption that it's impossible to determine the value of the private key. RSA encryption uses this formula:

$$C = M^e \pmod N$$



3.2.1 Code Key Generation:

```python
def generate_keys(username):
    private_key_file = f"{username}_private.pem"
    public_key_file = f"{username}_public.pem"

    if not (os.path.isfile(private_key_file) and os.path.isfile(public_key_file)):
        publicKey , privateKey = rsa.newkeys(1024) #1024 bytes long
        with open(private_key_file, "wb") as f:
            f.write(privateKey.save_pkcs1("PEM"))

        with open(public_key_file, "wb") as f:
            f.write(publicKey.save_pkcs1("PEM"))
```

3.2.2 Explanation:

File Naming: Depending on the username, the function defines the strings private_key_file and public_key_file. The file directories where the public and private keys will be kept are represented by these strings.

Next, it does a file check to see if the public and private key files are already present. It creates new keys and saves them if they don't already exist.

Key Generation: The function uses the rsa.newkeys(1024) method to create new RSA key pairs of 1024 bits if the files are not present. The public and private keys are contained in the tuple that this function returns.

File Writing: Next, it uses privateKey.save_pkcs1("PEM") to write the private key to the file after opening it in binary write mode ("wb"). The private key is saved in PEM format as a result.

The procedure is repeated with the public key, where the public key is written to the file in PEM format when the file is opened in binary write mode.

3.3 Password-based Authentication Module:

3.3.1 Register a new user

```python
def register_new(username, password):
    try:
        hashed_password = hashlib.sha256(password.encode()).hexdigest()
        with open("auth.txt", "a") as f:
            f.write(f"{username},{hashed_password}\n")
            print("Registered successfully")
        # Generate keys only if they don't already exist
        generate_keys(username)

    except Exception as e:
        print("Error:", e)
```

The register_new() function is responsible for adding a new user to the system into a file called “auth.txt” and saving the hashed password for the user in order to add more security , and call function of generation the public , private keys for him.

3.3.2 Sign in function

```python
import hashlib
import rsa
import os

def SignIn(username, password, filename):
    try:
        with open(filename, 'r') as file:
            for line in file:
                rows = line.strip().split(',')
                if rows[0] == username.strip():  # Strip leading/trailing whitespaces f
                    hashed_password = hashlib.sha256(password.encode()).hexdigest()
                    if rows[1] == hashed_password:
                        print("Signed in successfully")
                        return True
        print("Failed to login, check your name or password")
        return False
    except Exception as e:
        print("Error:", e)
        return False
```

The SignIn() function is responsible for authenticating a user into our system , It checks that this username and password exists in our file

Adding new user called test

```
Enter 1 to signIn or 2 to register
2
Enter your username: test
Enter your password: test123
Registered successfully
Enter 1 to signIn or 2 to register
```

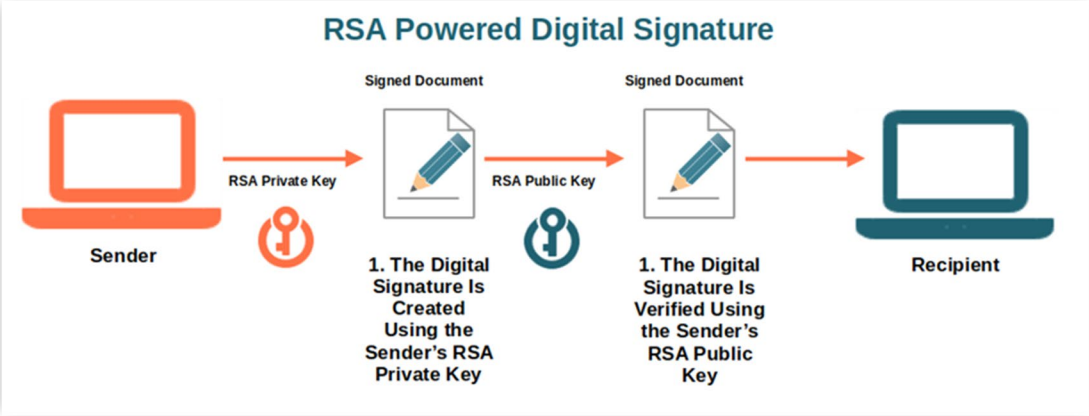
Our Auth.txt file and test username and hashed password are successfully added.

```
username,password
shimo,1b4f0e9851971998e732078544c96b36c3d01cedf7caa332359d6f1d83567014
aya,6b4e59de8a6e7508abda84cbda834f3e2ce8b8a5587d311ec81715c69a30a1d1
karen,fd2cad28fc6c1ab501785934cd44706a14ba4bf5a0ef2d54c9eed3f801197a64
Moataz,313ea3b7cb2e508ae8285e7e49615ef3b37f15eb487d1186d5ddfbd82c469779
rana,7ca8f7180eb9f506038586bbb08fe982306a9ffd37c744ec6868f6921f417c77
aya,6b4e59de8a6e7508abda84cbda834f3e2ce8b8a5587d311ec81715c69a30a1d1
shimo,8d05b6295cb7848dc3c8e24d12667f7f378ebd672c9c80965aaae7aa2fe2ebe0
aya,6b4e59de8a6e7508abda84cbda834f3e2ce8b8a5587d311ec81715c69a30a1d1
test,ecd71870d1963316a97e3ac3408c9835ad8cf0f3c1bc703527c30265534f75ae
```

3.4 Key Distribution and storage

Approach

1. We check that the user is signed in correctly using username and password.
2. Generate public and private keys associated with this user.
3. Allow the user to send a message by opening the message file.
4. Hash the message using SHA-256 technique.
5. Make a unique signature for each user.
6. At the receiver side, he will verify the message with the public key



3.5 Full integrated Code:

3.5.1 Sending a message

```
def Send_message():
    choice = input('Enter 1 to signIn or 2 to register \n')
    if choice == '1':
        username = input("Please enter your username :")
        password = input("Enter your password : ")
        isUser = SignIn(username, password, "auth.txt")
        if isUser:
            with open(f"{username}_private.pem", "rb") as f:
                private_key = rsa.PrivateKey.load_pkcs1(f.read())
            with open(f"{username}_public.pem", "rb") as f:
                public_key = rsa.PublicKey.load_pkcs1(f.read())

            receiver_username = input("Enter the receiver name : ")
            receiver_public_key_file = f"{receiver_username}_public.pem"
            if not os.path.isfile(receiver_public_key_file):
                print(f"Error: Receiver's public key not found. Ask
{receiver_username} to register.")
                return


             message = input("Enter the message you want to send : ")

            if not message:
                print('Please enter a message, it cannot be empty.')
                return

            message_bytes = message.encode()  # Convert message to bytes

            shared_secret_key = os.urandom(32)

            # Encrypt the message with AES using the shared secret key
            encrypted_message = encrypt_with_aes(message_bytes,shared_secret_key)

            # Encrypt the shared secret key with receiver's public key
            with open(receiver_public_key_file, "rb") as f:
                receiver_public_key = rsa.PublicKey.load_pkcs1(f.read())
            encrypted_shared_secret = rsa.encrypt(shared_secret_key,
receiver_public_key)

            # Add the message, private key for each user, and the hashed value in
the signature
            signature = rsa.sign(encrypted_message, private_key, "SHA-256")
```

```
            with open(f"{username}_signature", 'wb') as signature_file:
                signature_file.write(signature)
            with open("encrypted_msg", 'wb') as encrypted_msg:
                encrypted_msg.write(encrypted_message)

            print("Your signature on the message is successfully created")
            print("Encrypted Message:", encrypted_message)
            # Call receive_message function to decrypt and verify the message
            receive_message(receiver_username, username, encrypted_message,
encrypted_shared_secret, signature)
        else:
            print("Wrong username or password")

    elif choice == '2':
        username = input("Enter your username: ")
        password = input("Enter your password: ")
        register_new(username, password)
        Send_message()
```

Explanation

User Choice: Upon initialization, the function prompts the user to register or sign in.

Signing In: The user is asked for their username and password if they decide to sign in (input '1'). To find out if the user is registered, the SignIn() function is contacted. The function loads the user's private and public keys if the user is logged in.

Transmitting a Message
Next, the user inputs the message they want to send along with the username of the recipient.

Generate a shared secret key

A shared secret key is a piece of data that is only known by the persons engaging in secure communication. This often refers to the key of a symmetric cryptosystem. The shared secret key here is an array of randomly selected bytes so, this key will be used to encrypt the actual message with AES.
RSA is used to encrypt the shared secret key with the recipient's public key.

Encrypt the shared secret key with receiver's public key

```
with open(f"{receiver_username}_public.pem", "rb") as f:
 receiver_public_key = rsa.PublicKey.load_pkcs1(f.read())
encrypted_shared_secret = rsa.encrypt(shared_secret_key, receiver_public_key)
```

Illustration:

- Loading Receiver's Public Key
- Encrypting the Shared Secret Key : rsa.encrypt(shared_secret_key, receiver_public_key): Uses the receiver's public key to encrypt the shared secret key. This ensures that only the receiver, who possesses the corresponding private key, can decrypt and access the shared secret key.

The result, encrypted_shared_secret, is the ciphertext of the shared secret key, which can be safely transmitted along with the encrypted message.

Signing the Message

rsa.sign(encrypted_message, private_key, "SHA-256"): Uses RSA to sign the encrypted_message with the user's private_key and SHA-256 as the hashing algorithm.

The resulting signature ensures that the message came from the sender and was not altered.

Saving the signature in a file

This file can be transmitted along with the encrypted message to allow the recipient to verify the message's authenticity.

Why we sign the message? To achieve the following features

Authentication: Signing the encrypted message ensures that the recipient can verify the identity of the sender.

Integrity: It provides a guarantee that the encrypted message has not been altered during transmission. If the message is tampered with, the signature verification will fail.

Non-Repudiation: The sender cannot deny having sent the message, as only they have access to their private key used to create the signature.

3.5.2 Receiving a message

```
def receive_message(receiver_username, sender_username, encrypted_message,
encrypted_shared_secret, signature):
    # Load or generate receiver's private key
    private_key_file = f"{receiver_username}_private.pem"
    public_key_file = f"{receiver_username}_public.pem"
    if not (os.path.isfile(private_key_file) and
os.path.isfile(public_key_file)):
        print("Receiver's keys not found. Generating new keys...")
        generate_keys(receiver_username)

    with open(private_key_file, "rb") as f:
        receiver_private_key = rsa.PrivateKey.load_pkcs1(f.read())

    # Load sender's public key
    sender_public_key_file = f"{sender_username}_public.pem"
    if not os.path.isfile(sender_public_key_file):
        print(f"Error: Sender's public key not found. Ask {sender_username} to
register.")
        return
    with open(sender_public_key_file, "rb") as f:
        sender_public_key = rsa.PublicKey.load_pkcs1(f.read())
    # Decrypt the shared secret key using receiver's private key
    decrypted_shared_secret = rsa.decrypt(encrypted_shared_secret,
receiver_private_key)
    # Decrypt the message using the decrypted shared secret key
    decrypted_message = decrypt_with_aes(encrypted_message,
decrypted_shared_secret)
    # Verify the signature of the message using sender's public key
    try:
        rsa.verify(encrypted_message, signature, sender_public_key)
        print("Signature verification successful")
        print("Decrypted Message:", decrypted_message.decode())
    except rsa.VerificationError:
        print("Signature verification failed")
```

As The encrypted message, encrypted shared secret key, signed document, receiver's username, and sender's username are sent to the receive_message() method.

1. Load the receiver's private key from a file. This key is necessary to decrypt the shared secret key.
2. encrypted_shared_secret: The shared secret key, encrypted with the receiver's public key, was transmitted along with the message.
3. rsa.decrypt(encrypted_shared_secret, receiver_private_key): Decrypts the shared secret key using the receiver's private key.
4. We used the decrypted shared secret key to decrypt the encrypted message.
5. Using the sender's public key, it confirms the message's signature.
6. Error Resolution: If the signature verification is unsuccessful, it suggests that the message might have been altered.

Using a shared key in this approach

It plays a crucial role in secure communication between two parties for the following roles:

1. Confidentiality: The shared secret key ensures confidentiality by enabling encryption and decryption of messages exchanged between parties. Without the key, an attacker intercepting the communication would only see unintelligible ciphertext.

2. Symmetric Cryptography Efficiency: Symmetric cryptography algorithms, such as AES (Advanced Encryption Standard), are generally faster and require less computational resources compared to asymmetric algorithms like RSA. By using a shared secret key, symmetric encryption can be employed for encrypting the actual message, enhancing efficiency.

3. Key Exchange: The shared secret key needs to be securely exchanged between the communicating parties. This exchange typically occurs over a secure channel or through a secure key exchange protocol to prevent interception or tampering by adversaries.

4. Randomness and Entropy: The shared secret key should be generated using a secure random number generator to ensure unpredictability and resistance against brute-force attacks. High entropy keys are crucial for maintaining security.

However, because the same key is used to encrypt and decrypt secrets, proper key protecting, and distribution is critical. If the key is shared with unauthorized or unexpected receivers at any point during the information's life cycle, its security must be considered compromised.

4.0 Test Cases

Test Case 1 : Register a new user

```
Enter 1 to signIn or 2 to register
2
Enter your username: Aly
Enter your password: Aly123
Registered successfully
```

My auth.txt contain all our users

```
username,password
shimo,1b4f0e9851971998e732078544c96b36c3d01cedf7caa332359d6f1d83567014
aya,6b4e59de8a6e7508abda84cbda834f3e2ce8b8a5587d311ec81715c69a30a1d1
karen,fd2cad28fc6c1ab501785934cd44706a14ba4bf5a0ef2d54c9eed3f801197a64
Moataz,313ea3b7cb2e508ae8285e7e49615ef3b37f15eb487d1186d5ddfbd82c469779
test,ecd71870d1963316a97e3ac3408c9835ad8cf0f3c1bc703527c30265534f75ae
Aly,6aa9cf3957127f82581a48f6ab0c810f58f6ea7d18add654dd3c9d6cbcfdb1ff
```

Test Case 2 :Sign in with incorrect email or password

```
Enter 1 to signIn or 2 to register
1
Please enter your username :Aly
Enter your password : a
Failed to login, check your name or password
Wrong username or password
```

Testcase 3: Sending an empty message

```
Enter 1 to signIn or 2 to register
1
Please enter your username :rana
Enter your password : rana123
Signed in successfully
Enter the receiver name : aya
Enter the message you want to send :
Please enter a message, it cannot be empty.
```

Test case 4 : Sending a message to someone not registered on our system

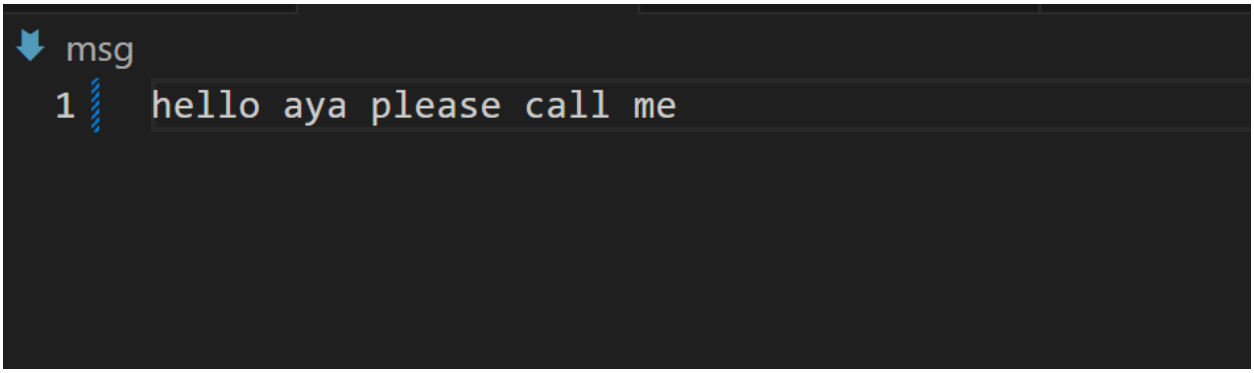
```
Enter 1 to signIn or 2 to register
1
Please enter your username :Aly
Enter your password : Aly123
Signed in successfully
enter the receiver name : Rana
Error: Receiver's public key not found. Ask Rana  to register.
PS D:\HPC_project\Cryptography_Application>
```

Test case 5 : Sending a message to someone registered on our system

```
Enter 1 to signIn or 2 to register
1
Please enter your username :Aly
Enter your password : Aly123
Signed in successfully
enter the receiver name : aya
enter the message you want to send : hello aya please call me
Your signature on the message is successfully created
Encrypted Message: b'P\xb7Q]\x9b1\xa6\xeflW\x92,\xef\x00\tx\xc9p)D\x15O?\x1f\xb6OK\xff\x0f\x81 .@=\
x05?\x98\xa8\xcb\xdb\xd5\xbf\x8e}\xb9\xfd\xab\xd7`\x15,4\xed\x19SG'
Signature verification successful
Decrypted Message: hello aya please call me
```

The receiver here is aya , and the message is successfully sent.

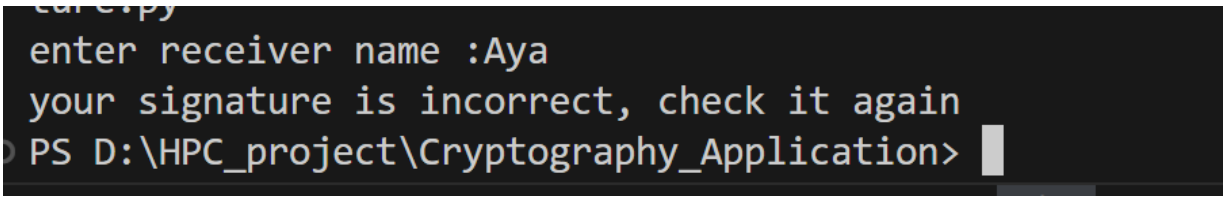
The message file



Signature file for the sender Aly

Test Case 6 Change in the message file , and try to verify the same signature file generated above

At the receiver side , it will say that the signature isnot correct



Test Case 7

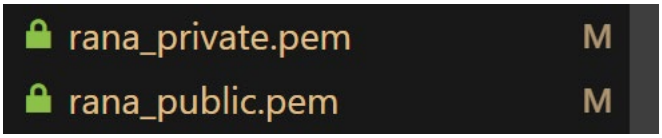
keep the same message and change sender to be Shaimaa and use my public key in verification, it refused because the signature created for Aly sending this specific message.

```python
import rsa
############## verify the signature at the receiver side ###############
input("enter receiver name :")
with open ("shimo_public.pem") as f :
    public_key = rsa.PublicKey.load_pkcs1(f.read())

with open ("aya_private.pem") as f :
    private_key = rsa.PrivateKey.load_pkcs1(f.read())

# get the value of the public key
message=open ('encrypted_msg','rb').read()
signature=open ("Aly_signature",'rb').read()

try:
    rsa.verify(message,signature,public_key)
    print("signature is correct , message sent correctly")

except:
    print("your signature is incorrect, check it again")
```

```
enter receiver name :Aya
your signature is incorrect, check it again
PS D:\HPC_project\Cryptography_Application>
```

Test Case 8 at registering a new user , new public and private keys are generated and If I registered again with the same data , it will not generate new keys.

```
Enter 1 to signIn or 2 to register
2
Enter your username: rana
Enter your password: rana123
Registered successfully
```



Test Case 9: Change the decryption of message to be with the receiver private key not the shared one , error will be generated as this is a wrong decryption key.

```
Please enter your username :rana
Enter your password : rana123
Signed in successfully
Enter the receiver name : aya
Enter the message you want to send : hi aya
Your signature on the message is successfully created
Encrypted Message: b'\xde\xa8\xbe%\xe5)<\xe4\xc1\xc9!y\x80g\xe2D\xad\xac\xf0\xb4\xe8\xf14\xe2rN\x9c
!\x1c\x82\xbe\xc2\xd97\xf1\xd5\xe4\xf2'
Traceback (most recent call last):
  File "d:\HPC_project\Cryptography_Application\Phase4\final_integration.py", line 156, in <module>
    Send_message()
  File "d:\HPC_project\Cryptography_Application\Phase4\final_integration.py", line 111, in Send_mes
sage
    receive_message(receiver_username, username, encrypted_message, encrypted_shared_secret, signat
ure)
  File "d:\HPC_project\Cryptography_Application\Phase4\final_integration.py", line 146, in receive_
message
    decrypted_message = decrypt_with_aes(encrypted_message, receiver_private_key)
```

Conclusion

This method provides a robust and efficient way to securely transmit encrypted messages. The combination of AES and RSA leverages the best of both symmetric and asymmetric encryption, ensuring both the security and efficiency of the encryption process. This hybrid encryption method is widely used in modern cryptographic systems due to the following :

1. **Security**:
    - **Confidentiality**: AES provides fast and secure encryption of the actual message. Using a random key for each message increases security.
    - **Integrity**: AES in EAX mode ensures the message has not been tampered with.
    - **Key Security**: The shared secret key is protected by RSA encryption, ensuring only the intended recipient can decrypt and use it.
2. **Efficiency**:
    - **Performance**: AES is highly efficient for encrypting large messages.
    - **Hybrid Approach**: By encrypting only the small shared secret key with RSA, we avoid the inefficiency of using RSA on large data.
3. **Flexibility**:
    - The approach allows for the secure transmission of any message, regardless of size, by combining the strengths of both AES and RSA.
    - It is scalable and can be used in various scenarios, such as secure email, file transfer, and messaging applications.

5.0 Questions and Answers

1. How are the different modules integrated into the Secure Communication Suite?

The Secure Communication Suite integrates different modules to ensure a comprehensive security solution for communication. These modules were discussed with more details in the Key Features section above and it generally includes:

- Block Cipher Module: Utilizes symmetric encryption methods like DES and AES for encrypting and decoding data, ensuring confidentiality.
- Public Key Cryptosystem Module: Implements asymmetric encryption techniques such as ECC and RSA, enabling secure key exchange and communication between parties.
- Hashing Module: Creates unique checksums for data using techniques like MD5 or SHA-256 to confirm data integrity.
- Key Management Module: Provides secure techniques for generating, distributing, and storing keys, ensuring keys are protected from unauthorized access.
- Authentication Module: Implements authentication procedures to verify user identity, preventing unauthorized access to sensitive data.
- Internet Services Security Module: Utilizes cryptographic modules to secure data for internet services, ensuring confidentiality and integrity.

2. What types of tests were conducted on the suite?

We Conducted various tests on the Suite to ensure its functionality and security. These tests were discussed with more details in the Test Cases section and it generally includes:

- Registering a new user to validate the user registration process.
- Signing in with correct and incorrect credentials to verify authentication mechanisms.
- Sending messages to test message encryption, signature generation, and decryption.
- Testing edge cases such as sending empty messages, sending messages to unregistered users, and verifying message integrity.
- Testing key generation, distribution, and storage to ensure the security of key management operations.

3. How does the suite secure internet services?

The Secure Communication Suite employs cryptographic methods and security protocols to safeguard internet services. By integrating modules for encryption, authentication, and key management, the suite ensures the authenticity, integrity, and confidentiality of communications over the internet. It uses symmetric and asymmetric encryption techniques to encrypt data and securely exchange keys between parties (using shared secret key). Additionally, hashing is utilized to verify data integrity, while authentication procedures prevent unauthorized access to the suite's capabilities. Overall, the suite provides a comprehensive solution for protecting internet services against surveillance and tampering.

GitHub link

https://github.com/Aya-gamal2211/Cryptography_Application