TERMINAL

Subject: Information Security Lab

Submitted To: Mam Ambreen Gul

Submitted By: Muhammad Hassan

Registration Number: Sp24-Bse-008

Date: 15/12/2025

Question 1:

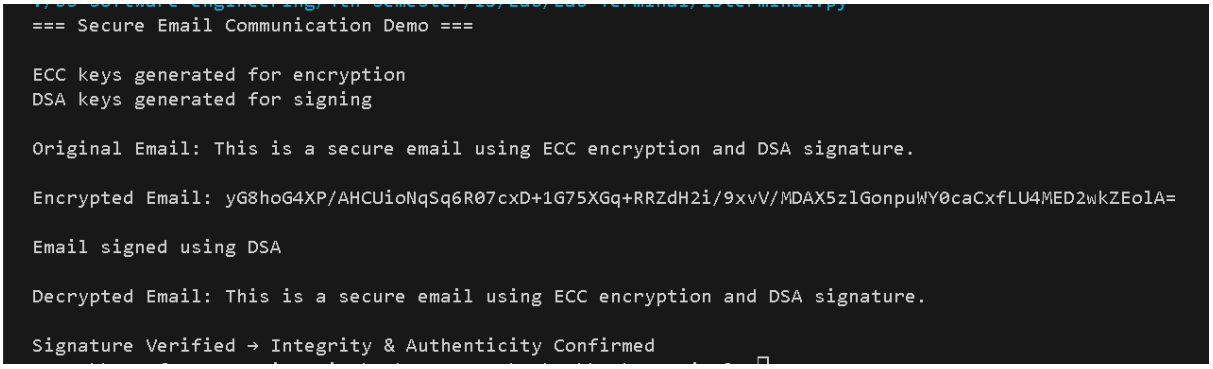
Code:

```
# ==============================
# Secure Email System: ECC + DSA
# ==============================

from Crypto.PublicKey import ECC, DSA
from Crypto.Signature import DSS
from Crypto.Hash import SHA1
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Protocol.KDF import HKDF
import base64

# ================================================
# 1️⃣ ECC KEY GENERATION (For Encryption/Decryption)
# ================================================

def generate_ecc_keys():
    """
    Generate ECC key pair using P-256 curve.
    Private key: Used to decrypt messages
    Public key: Used to encrypt messages
    """
    private_key = ECC.generate(curve='P-256')
    public_key = private_key.public_key()
    return private_key, public_key

# ================================================
# 2️⃣ ECC ENCRYPTION & DECRYPTION (Hybrid ECC + AES)
# ================================================

def ecc_encrypt(message, receiver_public_key, sender_private_key):
    """
    Encrypt message using receiver's ECC public key
    - Uses ECC to derive shared secret
    - AES GCM for actual encryption
    """
    shared_point = receiver_public_key.pointQ * sender_private_key.d
    shared_secret = int(shared_point.x).to_bytes(32, 'big')
    aes_key = HKDF(shared_secret, 32, b'', SHA1)

    cipher = AES.new(aes_key, AES.MODE_GCM)
```

```
    ciphertext, tag = cipher.encrypt_and_digest(message.encode())
    return cipher.nonce, ciphertext, tag

def ecc_decrypt(nonce, ciphertext, tag, receiver_private_key,
sender_public_key):
    """
    Decrypt message using ECC private key
    """
    shared_point = sender_public_key.pointQ * receiver_private_key.d
    shared_secret = int(shared_point.x).to_bytes(32, 'big')
    aes_key = HKDF(shared_secret, 32, b'', SHA1)

    cipher = AES.new(aes_key, AES.MODE_GCM, nonce=nonce)
    plaintext = cipher.decrypt_and_verify(ciphertext, tag)
    return plaintext.decode()

# ================================================
# 3️⃣ DSA KEY GENERATION (For Signing/Verification)
# ================================================

def generate_dsa_keys():
    """
    Generate DSA key pair for signing emails
    """
    private_key = DSA.generate(2048)
    public_key = private_key.publickey()
    return private_key, public_key

# ================================================
# 4️⃣ SIGNING & VERIFYING EMAIL (DSA)
# ================================================

def sign_email(message, dsa_private_key):
    """
    Sign the email message using DSA private key
    """
    hash_obj = SHA1.new(message.encode())
    signer = DSS.new(dsa_private_key, 'fips-186-3')
    signature = signer.sign(hash_obj)
    return signature

def verify_email(message, signature, dsa_public_key):
    """
    Verify email signature using DSA public key
    """
    hash_obj = SHA1.new(message.encode())
    verifier = DSS.new(dsa_public_key, 'fips-186-3')
    try:
```

```
        verifier.verify(hash_obj, signature)
        return True
    except ValueError:
        return False

# ================================================
# 5️⃣ MAIN FUNCTION: DEMO SECURE EMAIL
# ================================================

def main():
    print("=== Secure Email Communication Demo ===\n")

    # Step 1: Generate Keys
    ecc_private, ecc_public = generate_ecc_keys()
    dsa_private, dsa_public = generate_dsa_keys()
    print("ECC keys generated for encryption")
    print("DSA keys generated for signing\n")

    # Step 2: Compose email
    email_message = "This is a secure email using ECC encryption and DSA
signature."
    print("Original Email:", email_message, "\n")

    # Step 3: Encrypt email
    nonce, ciphertext, tag = ecc_encrypt(email_message, ecc_public,
ecc_private)
    print("Encrypted Email:", base64.b64encode(ciphertext).decode(), "\n")

    # Step 4: Sign email
    signature = sign_email(email_message, dsa_private)
    print("Email signed using DSA\n")

    # Step 5: Decrypt email
    decrypted_message = ecc_decrypt(nonce, ciphertext, tag, ecc_private,
ecc_public)
    print("Decrypted Email:", decrypted_message, "\n")

    # Step 6: Verify signature
    if verify_email(decrypted_message, signature, dsa_public):
        print("Signature Verified → Integrity & Authenticity Confirmed")
    else:
        print("Signature Invalid → Message may be tampered")

# Run the program
if __name__ == "__main__":
    main()
```

Output:



Working:

The system first generates ECC keys to securely encrypt and decrypt the email message.
A shared secret is created using ECC and converted into an AES key to protect message confidentiality.
DSA is then used to sign the message so the receiver can verify integrity and authenticity.

Function:

- Functions Used in Secure Email Communication System
- ec generate private key generates the ECC private key used for secure encryption and decryption
- public key generates the ECC public key that is shared for key exchange
- private bytes converts the ECC private key into PEM format for storage or display
- public bytes converts the ECC public key into PEM format for sharing
- dsa generate private key generates the DSA private key used for digital signing
- exchange creates a shared secret using elliptic curve key agreement
- hkdf derive generates a secure AES key from the shared secret
- urandom generates a random initialization vector for AES encryption
- cipher creates an AES encryption and decryption object
- encryptor encrypts the email message using the AES key
- decryptor decrypts the encrypted email back into original message
- sign creates a digital signature of the email using DSA private key
- verify verifies the digital signature using DSA public key

(Qno2)

Project: "Secure Text Encryption System Using Vernam Cipher (One-Time Pad)"

A) Justification of method:

The **Vernam Cipher (One-Time Pad)** is highly suitable for this project because it provides **perfect secrecy** when implemented correctly. In our project:

- A **random key** is generated using the generate_key() function.
- The key is **equal in length** to the plaintext.
- Each character of plaintext is encrypted with a **corresponding key character** using **mod 26 addition**.
- The same key is required for decryption.

Unlike other encryption methods (such as Caesar or Vigenère cipher), the Vernam Cipher has **no repeating patterns**, making it **immune to frequency analysis attacks**.

Additionally:

- It is **simple to implement**
- Very **fast**
- Requires **no complex mathematics**

Therefore, for a **text-based encryption project**, Vernam Cipher is **better than many classical methods** and clearly demonstrates the core principles of information security.

Confidentiality is fully ensured

B) Weakness: Insecure Key Management

The main weakness in the current system is **key handling and reuse**.

- The key is generated randomly, but:
  - It is **stored or displayed openly**
  - There is **no secure key exchange mechanism**
- If the **same key is reused** for multiple messages, the system becomes vulnerable.

How an attacker could misuse it

If an attacker:

- Gains access to the key
- Or observes two ciphertexts encrypted with the same key

Then:

- The attacker can apply **known-plaintext attacks**
- Or recover the original message using simple calculations

This completely breaks the security of the Vernam Cipher.

Key reuse destroys perfect secrecy

c): Improvement: Secure Key Distribution and Input Validation

A realistic improvement would be:

1. Secure Key Sharing

- Do not display the key openly
- Share the key using a **secure channel** (e.g., password-protected file or encrypted transfer)

2. Input Validation

- Restrict plaintext to **uppercase alphabetic characters (A–Z)**
- Reject numbers or spaces like "123 " which your current code allows

3. Automatic One-Time Use

- Ensure the key is **used only once**
- Generate a **new key for every message**

How it works

- This prevents key reuse
- Maintains the **One-Time Pad principle**
- Preserves **perfect secrecy**

Security becomes much stronger
Attacks become impractical