Lab 3: Public Key Cryptography Implementation

- In this lab I will learn about asymmetric key (public key) cryptography. I will implement the Diffie-Hellman Key Exchange protocol and RSA encryption scheme.
    - I will explore the properties of these schemes to see how naive approaches to these implementations can lead to insecurity

Task 1: Implementing the Diffie-Hellman Key Exchange Protocol

Initial Setup

1. **Create Public Parameters:**
    - The two people sending & receiving encrypted messages (Alice & Bob) must agree on a large prime number (q) and a base (alpha)
    - These parameters are made public and can be transferred openly.

Key Creation Process

2. **Private and Public Keys:**
    - **Private Key:** Each person must pick a random 2048 bit number (X_A) as their private key, which will be kept a secret.
        - This key must not leave the machine it was created on.
    - **Public Key:** Each person must also have a public key that is available to the public. This will be computed using alpha^X_sender * mod q

```
def generate_private_and_public(alpha, q):
  private_key = random.randint(1, 2**2048)
  public_key = pow(alpha, private_key, q)
  return {"private": private_key, "public": public_key}
```

Generating the Shared Secret

3. Each person must generate their shared key which should be equal to each others.
    - Compute the shared secret by raising the other party's public key to the power of their own private * modulo q

```
def compute_shared_secret(other_public_key, private_key, q):
  """Compute the shared secret using the other's public key and your private key."""
  return power_modulo(other_public_key, private_key, q)
```

Deriving a Symmetric Key

4. Both parties must also hash the shared secret using SHA256 and truncate the output ti 16 bytes, so it can be used to AES-CBC encrypt the messages sent.

```
def derive_symmetric_key(shared_secret):
  """Hash the shared secret and truncate to derive the AES key."""
  hasher = sha256()
  hasher.update(shared_secret.to_bytes((shared_secret.bit_length() + 7) // 8, byteorder='big'))
  return hasher.digest()[:16]
```

Message Encryption and Transmission

5. Using the symmetric key to encrypt messages with AES in CBC mode

```
 ciphertext = encrypt_with_cbc_mode(plaintext.encode('utf-8'), sender_symmetric_key, iv)
decrypted_text = decrypt_with_cbc_mode(ciphertext, sender_symmetric_key, iv)
```

Task 2: Implement Man-in-the-Middle Attack

- In this task I will attempt to introduce an attacker into the previous implementation of the Diffie-Hellman Key Exchange process. This attacker will intercept and modify the public keys tricking both the sender and the receiver.
    - Instead of the sender and receiver exchanging public keys, the attacker will send the modified values to each of them
        - Meaning the attacker will send q to each of them.
    - The attacker will also compute the shared secret, since they send q to both people. Shared secret (s) = q^private_key mod q
    - The sender and receiver will not notice anything different, since they will continue to encrypt and decrypt messages using the shared key computed.
    - I believe this attack can be very damaging for this reason, since the attacker compromised their shared secret without the sender or the receiver noticing anything.

Implementation:

- Implementation will be similar to Task 1, but I will show the additional steps where an attacker will compromise the shared secret.

1. Attacker intervention: The attacker (Mallory) intercepts the public key and send q to both the sender and the receiver

```
 # Mallory's intervention - sending q instead of the public keys
def mallory_send_q_instead(public_key):
    return q
```

2. Computation of Shared Secret: The sender and the receiver use `q` to compute the shared secret, believing that q was the public key of the other party.

```
def compute_shared_secret(other_public_key, private_key, q):
return power_modulo(other_public_key, private_key, q)
```

3. Encryption and Decryption of messages: The sender and the receiver use this shared secret to create the symmetric key that is used to encrypt and decrypt messages using CBC

```
 # Both Alice and Bob compute the shared secret using the modified public keys (which is now q)
alice_shared_secret = compute_shared_secret(alice_believes_bobs_public, alice['private'], q)
bob_shared_secret = compute_shared_secret(bob_believes_alices_public, bob['private'], q)

# Derive symmetric keys based on the compromised shared secrets
alice_symmetric_key = derive_symmetric_key(alice_shared_secret)
bob_symmetric_key = derive_symmetric_key(bob_shared_secret)
iv = random.randbytes(16)

# Alice sends a message to Bob
alice_msg = "Hi, Bob!"
alice_ciphertext = encrypt_with_cbc_mode(alice_msg.encode('utf-8'), alice_symmetric_key, iv)
```

4. Mallory Can also Decrypt Messages: Since, the attacker (Mallory) knows the shared secret, she can derive the key and decrypt the message

```
 mallorys_key = derive_symmetric_key(0)  # or derive_symmetric_key(1), depending on the value of q^private_key mod q
mallory_decrypted_alice = decrypt_with_cbc_mode(alice_ciphertext, mallorys_key, iv)
```

Task 3

- In this task, I will implement the textbook algorithm of RSA. Her are the two core components of RSA that I need to create:
  - **Key Generation:** This is the majority of the work. Each entity `A` will do the following steps.
  1. Generate two large random (and distinct) primes p & q. Should be approximately the same size.

```
 p = getPrime(bit_length)
q = getPrime(bit_length)
```

  2. Compute n = p *q and ø = (p-1)*(q-1)

```
 n = p * q
phi_n = (p - 1) * (q - 1)
```

  3. Select a random int, `e`, that has the following requirements:
     - 1 < e < ø
     - gcd(e, ø) = 1

```
  while gcd(e, phi_n) != 1:
    p = getPrime(bit_length)
    q = getPrime(bit_length)
    n = p * q
    phi_n = (p - 1) * (q - 1)
```

  4. Use the `extended Euclidean algorithm` to compute unique int, `d`, that has the following requirements:
     - 1 < d < ø
     - e*d = 1*mod ø

```
def extended_euclidean(a, b):
 if b == 0:
     return a, 1, 0

 x2, x1 = 1, 0
 y2, y1 = 0, 1

 while b > 0:
     q = a // b
     r = a - q * b
     x = x2 - q * x1
     y = y2 - q * y1

     a, b = b, r
     x2, x1 = x1, x
     y2, y1 = y1, y

 return a, x2, y2
```

```
def mod_inverse(e, phi):
  gcd, x, _ = extended_euclidean(e, phi)
  if gcd != 1:
      raise Exception('Modular inverse does not exist')
  else:
      return x % phi
```

```
    d = mod_inverse(e, phi_n)
```

5. Entity A's public key is `(n, e)` and A's private key is `d`

```
def generate_rsa_keys(bit_length, e=65537):
 p = getPrime(bit_length)
 q = getPrime(bit_length)
 if p == q:
     q = getPrime(bit_length)  # Ensure p and q are distinct
 n = p * q
 phi_n = (p - 1) * (q - 1)

 while gcd(e, phi_n) != 1:
     p = getPrime(bit_length)
     q = getPrime(bit_length)
     n = p * q
     phi_n = (p - 1) * (q - 1)

 d = mod_inverse(e, phi_n)
 return ((n, e), d)
```

- **RSA Public-Key Encryption:** Entity B will encrypt a message (m) for A

1. **Encryption:** B does the following a. Obtain A's public key `(n, e)`

   b. Convert message (m) to an integer (if necessary) between the interval [0, n-1]

   c. Compute ciphertext (`c`)where `c = m^e mod n`

   d. Send the ciphertext (`c`) to A

```
def rsa_encrypt(public_key, plaintext):
 n, e = public_key
 # Convert plaintext string to an integer
 m = int.from_bytes(plaintext.encode('utf-8'), 'big')
 if m >= n:
     raise ValueError("Plaintext too long for key size")
 c = pow(m, e, n)
 return c
```

2. **Decryption:** To recover the plaintext message (m), we need to use the private key `d` and do the following: a. `m = c^d mod n`

```
def rsa_decrypt(private_key, public_key, ciphertext):
 n, e = public_key
 d = private_key
 m = pow(ciphertext, d, n)
 plaintext = m.to_bytes((m.bit_length() + 7) // 8, 'big').decode('utf-8')
 return plaintext
```

Malleability Attack

- In this task, I will implement an attack where I will intercept and modify the the ciphertext. This attack resulted in the receiver getting an altered message and the attacker decrypting the actual message.

```
def malleability():
    bit_length = 1024
    public_key, private_key = generate_rsa_keys(bit_length)
    message = 123456

    # Alice encrypts a message
    c = rsa_encrypt(public_key, message)

    # Mallory intercepts and modifies the ciphertext
    s = 3
    c_prime = (c * pow(s, public_key[1], public_key[0])) % public_key[0]

    # Alice decrypts the modified ciphertext
    m_prime = rsa_decrypt(private_key, public_key, c_prime)

    # Mallory calculates the original message
    recovered_message = (m_prime * inverse(s, public_key[0])) % public_key[0]

    print("Original message:", message)
    print("Decrypted modified message:", m_prime)
    print("Mallory's recovered message:", recovered_message)
```

Forging Signature

- In this attack I also used Malleability to affect the signature schemes to forge a valid signature.

```
def rsa_signature(message, private_key, n):
    """Create an RSA signature using the private key."""
    return pow(message, private_key, n)

def forge_signature(sig1, sig2, n):
    """Forge a new signature by multiplying two signatures."""
    return (sig1 * sig2) % n

def signature_forgery():
    # RSA parameters
    n = 3233  # public modulus
    d = 2753  # private key
    e = 17    # public exponent

    # Two messages and their RSA signatures
    m1 = 123
    m2 = 456
    sig_m1 = rsa_signature(m1, d, n)
    sig_m2 = rsa_signature(m2, d, n)

    # Mallory forges a signature for m3 = m1 * m2
    m3 = m1 * m2
    forged_sig_m3 = forge_signature(sig_m1, sig_m2, n)

    print("Original Signature for m1:", sig_m1)
    print("Original Signature for m2:", sig_m2)
    print("Forged Signature for m3:", forged_sig_m3)
    print("Verification of forged signature:", pow(forged_sig_m3, e, n) == m3 % n)
```

QnA

1. For task 1, how hard would it be for an adversary to solve the Diffie-Hellman Problem (DHP) given these parameters? What strategy might the adversary take?

    - **Answer:** Solving the Diffie-Hellman Problem with the given parameters, q & alpha, would be extremely difficult due to the large sizes of the prime numbers. Having these numbers over 300 digits long means that the discrete logarithm problem it needs to solve to be attacked would take a lot of computational power.

      A basic brute force attack to try and break the discrete log problem would not be an approach an adversary would take. Instead they might use a man-in-the-middle attack, since the public keys are easily obtainable the attacker can intercept the public keys sent by Alice and Bob to each other. Without any sort of authorization, Bob & Alice would be communication through an attacker (Mallory) and not even know it. In the man in the middle attack, Mallory would send their public key to Bob pretending it came from Alice, and vice versa. In doing this the attacker can now generate a shared secret key that can be used for decryption and re-encryption.

2. For task 1, would the same strategy used for the tiny parameters work for the large values of q and alpha? Why or why not?

    - **Answer:** The effectiveness of the Diffie-Hellman Key exchange is dependent on the parameter sizes for q & alpha. Using tiny parameters would make it easier to do a brute force attack knowing that the discrete log problem could be solved more easily do to it needing less computational power.

    The same strategy of man-in-the-middle would work the same, since it does not rely on the size of q or alpha. It instead relies on the lack of authentication. However, it might now be easier to just do a brute force attack depending on how much smaller the parameters became.

3. For task 2, why were these attacks possible? What is necessary to prevent it?

    - **Answer:** These attacks were possible because of 2 main reasons:

        1. No authentication of the public keys. This allowed an attacker to intercept the public keys and replace them with any other value without the sender or the receiver finding out.
        2. No integrity checks. This allowed public keys to be altered while the sender and receiver were exchanging messages.

    - **How to Prevent:**

        1. Public Key authentication using digital signatures where each party signs their public key with a private key issued through a trusted 3rd party such as certificate authority (CA)

4. For task 3 part 1, while it's very common for many people to use the same value for e in their key (common values are 3, 7, 216+1), it is very bad if two people use the same RSA modulus n. Briefly describe why this is, and what the ramifications are

    - **Answer:** Using the same modulus n can cause problems because of the following reasons:

        1. The main reason why RSA is difficult to break is due to how difficult it is factorize n back into p & q. However, this is not the case if they use the same modulus, since it can be easier to factorize.
        2. Easy to factorize: Computing the GCD of the same modulus could make it a lot easier to find p & q, especially when using the Euclidean algorithm.

    - **Ramifications:** Here are the following ramifications of sharing the same modulus.

        1. Private kets can be compromised, since an attacker could decrypt ø(n) = (p-1)(q-1)
        2. With access to private keys, an attacker could also forge digital signatures that could compromise integrity checks.