**LA 2 WRITEUP**

**Name: Paras Wane**

**PRN : 21510041**

**Batch: B 2**

ASSIGNMENT 7

#### 1. ****Key Generation:****

* **Step 1:** Choose two large prime numbers, ppp and qqq.
* **Step 2:** Compute n=p×qn = p \times qn=p×q and ϕ(n)=(p−1)(q−1)\phi(n) = (p-1)(q-1)ϕ(n)=(p−1)(q−1).
* **Step 3:** Select the public exponent eee such that 1<e<ϕ(n)1 < e < \phi(n)1<e<ϕ(n), and eee is co-prime to ϕ(n)\phi(n)ϕ(n).
* **Step 4:** Compute the private exponent ddd, which is the modular multiplicative inverse of eee modulo ϕ(n)\phi(n)ϕ(n), i.e., d×e≡1 (mod ϕ(n))d \times e \equiv 1 \ (\text{mod} \ \phi(n))d×e≡1 (mod ϕ(n)).

#### 2. ****Encryption:****

* **Formula:** c≡me (mod n)c \equiv m^e \ (\text{mod} \ n)c≡me (mod n)
  + Where:
    - mmm is the plaintext message.
    - ccc is the ciphertext.
    - eee is the public key exponent.
    - nnn is the product of the two primes ppp and qqq.

#### 3. ****Decryption:****

* **Formula:** m≡cd (mod n)m \equiv c^d \ (\text{mod} \ n)m≡cd (mod n)
  + Where:
    - mmm is the decrypted message (plaintext).
    - ccc is the ciphertext.
    - ddd is the private key exponent.

#### 4. ****Digital Signatures:****

* **Signing:**
  + **Formula:** s≡md (mod n)s \equiv m^d \ (\text{mod} \ n)s≡md (mod n)
  + Where:
    - sss is the digital signature.
    - mmm is the message.
    - ddd is the private key.
* **Verification:**
  + **Formula:** m≡se (mod n)m \equiv s^e \ (\text{mod} \ n)m≡se (mod n)
  + Where:
    - mmm is the original message.
    - sss is the signature.
    - eee is the public key.

### ****Applications:****

* **Secure Communications:** Used in protocols such as SSL/TLS.
* **Digital Signatures:** Ensures the authenticity and integrity of messages.

#### ****Conclusion:****

RSA is fundamental for ensuring data security in various digital applications, particularly for secure communications and digital signatures.

CODE:

#include <iostream>

#include <cmath>

#include <cstdlib>

#include <ctime>

using namespace std;

// Function to calculate the greatest common divisor (GCD)

int gcd(int a, int b) {

    while (b != 0) {

        int temp = b;

        b = a % b;

        a = temp;

    }

    return a;

}

// Function to perform modular exponentiation (x^y % p)

long long mod\_exp(long long x, long long y, long long p) {

    long long result = 1;

    x = x % p;

    while (y > 0) {

        if (y % 2 == 1)

            result = (result \* x) % p;

        y = y / 2;

        x = (x \* x) % p;

    }

    return result;

}

// Function to find the multiplicative inverse of e under mod phi

int mod\_inverse(int e, int phi) {

    for (int d = 1; d < phi; d++) {

        if ((e \* d) % phi == 1)

            return d;

    }

    return -1;

}

// Function to check if a number is prime

bool is\_prime(int num) {

    if (num < 2) return false;

    for (int i = 2; i <= sqrt(num); i++) {

        if (num % i == 0) return false;

    }

    return true;

}

int main() {

    srand(time(0));

    // Step 1: Choose two large prime numbers p and q

    int p, q;

    do {

        p = rand() % 100 + 100; // Random number between 100 and 200

    } while (!is\_prime(p));

    do {

        q = rand() % 100 + 100; // Random number between 100 and 200

    } while (!is\_prime(q));

    // Step 2: Compute n = p \* q

    int n = p \* q;

    // Step 3: Compute the totient phi = (p - 1) \* (q - 1)

    int phi = (p - 1) \* (q - 1);

    // Step 4: Choose an integer e such that 1 < e < phi and gcd(e, phi) = 1

    int e;

    do {

        e = rand() % phi;

    } while (gcd(e, phi) != 1);

    // Step 5: Compute d such that (d \* e) % phi = 1 (d is the modular inverse of e)

    int d = mod\_inverse(e, phi);

    // Display public and private keys

    cout << "Public Key: {" << e << ", " << n << "}\n";

    cout << "Private Key: {" << d << ", " << n << "}\n";

    // Step 6: Encryption

    int message;

    cout << "Enter the message (an integer): ";

    cin >> message;

    long long encrypted\_message = mod\_exp(message, e, n);

    cout << "Encrypted Message: " << encrypted\_message << "\n";

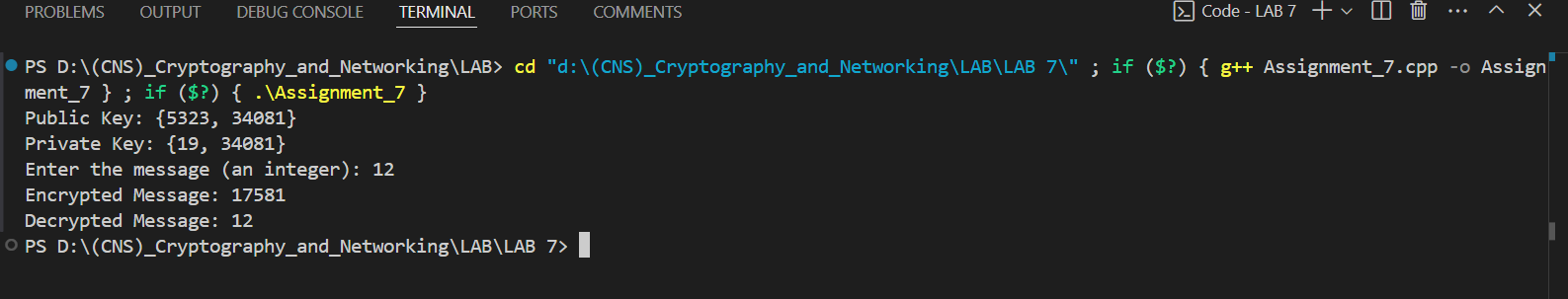
    // Step 7: Decryption

    long long decrypted\_message = mod\_exp(encrypted\_message, d, n);

    cout << "Decrypted Message: " << decrypted\_message << "\n";

    return 0;

}



**ASSIGNMENT 8**

**Implement the Diffie-Hellman Key Exchange algorithm for a given problem.**

**Introduction:** The Diffie-Hellman Key Exchange (DHKE) is a method that allows two parties to securely share a secret key over a public channel without prior secret sharing.

**Key Concepts:**

* **Public Parameters:** A large prime number ppp and a primitive root ggg (also known as a generator).
* **Private Keys:** Each party selects a private key (secret) that is kept confidential.

**Algorithm Steps:**

1. **Initialization:**
   * Two parties, Alice and Bob, agree on a prime number ppp and a base ggg.
2. **Key Generation:**
   * Alice selects a private key aaa and computes A=gamod  pA = g^a \mod pA=gamodp.
   * Bob selects a private key bbb and computes B=gbmod  pB = g^b \mod pB=gbmodp.
3. **Exchange:**
   * Alice sends AAA to Bob, and Bob sends BBB to Alice.
4. **Shared Secret Computation:**
   * Alice computes the shared secret: SA=Bamod  pS\_A = B^a \mod pSA​=Bamodp.
   * Bob computes the shared secret: SB=Abmod  pS\_B = A^b \mod pSB​=Abmodp.
   * Both will arrive at the same shared secret SSS.

**Security Basis:** The security of DHKE lies in the difficulty of solving the discrete logarithm problem, making it challenging for an eavesdropper to determine the shared secret from the exchanged values.

**Applications:**

* Used in secure communications (e.g., SSL/TLS) and to establish secure keys for symmetric encryption.

**Conclusion:** The Diffie-Hellman Key Exchange is a foundational technique for secure key exchange, enabling secure communication over untrusted networks.

**CODE:**

#include <iostream>

#include <cmath>

using namespace std;

// Function to perform modular exponentiation (a^b % mod)

long long mod\_exp(long long base, long long exp, long long mod) {

    long long result = 1;

    base = base % mod;

    while (exp > 0) {

        if (exp % 2 == 1)

            result = (result \* base) % mod;

        exp = exp >> 1;

        base = (base \* base) % mod;

    }

    return result;

}

int main() {

    // Step 1: Publicly agree on a large prime number (p) and a primitive root (g)

    long long p = 23;  // Example prime number

    long long g = 5;   // Example primitive root

    cout << "Publicly Shared Prime (p): " << p << endl;

    cout << "Publicly Shared Primitive Root (g): " << g << endl;

    // Step 2: Each user chooses a private key

    long long a;  // Private key for User A

    long long b;  // Private key for User B

    cout << "Enter the private key for User A: ";

    cin >> a;

    cout << "Enter the private key for User B: ";

    cin >> b;

    // Step 3: Each user calculates their public value

    long long A = mod\_exp(g, a, p);  // User A's public value

    long long B = mod\_exp(g, b, p);  // User B's public value

    cout << "User A's Public Value (A): " << A << endl;

    cout << "User B's Public Value (B): " << B << endl;

    // Step 4: Each user computes the shared secret

    long long shared\_secret\_A = mod\_exp(B, a, p);  // User A's computed shared secret

    long long shared\_secret\_B = mod\_exp(A, b, p);  // User B's computed shared secret

    cout << "User A's Computed Shared Secret: " << shared\_secret\_A << endl;

    cout << "User B's Computed Shared Secret: " << shared\_secret\_B << endl;

    // If both computed shared secrets match, the exchange is successful

    if (shared\_secret\_A == shared\_secret\_B) {

        cout << "Shared secret successfully established: " << shared\_secret\_A << endl;

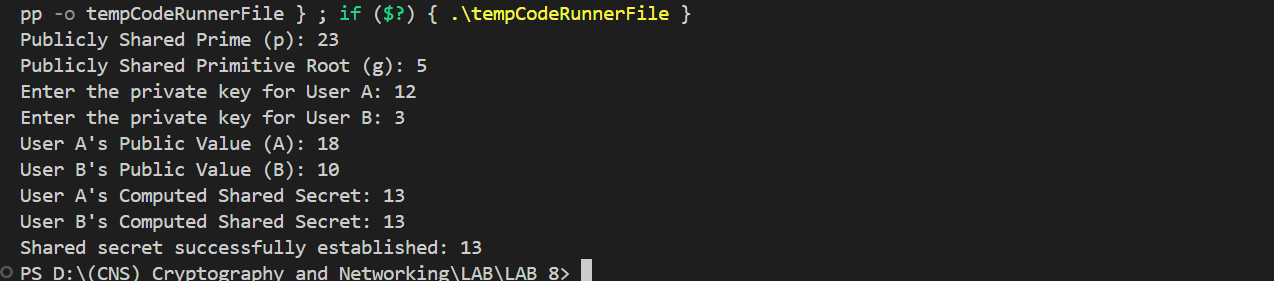
    } else {

        cout << "Error in establishing the shared secret." << endl;

    }

    return 0;

}

****

**ASSIGNMENT 9**

**Calculate the message digest of a text using the SHA-1 algorithm**.

**SHA-1 Message Digest: Overview**

**Introduction:** SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function designed to produce a fixed-size (160-bit) message digest from an input of arbitrary size. It is commonly used in various security applications and protocols, including TLS and SSL, PGP, and SSH.

**Key Characteristics:**

* **Deterministic:** The same input will always produce the same output.
* **Fixed Output Size:** Regardless of input size, the output is always 160 bits.
* **Fast Computation:** SHA-1 is designed for efficient processing.
* **Pre-image Resistance:** Given a hash output, it should be computationally infeasible to find any input that hashes to that output.
* **Collision Resistance:** It should be challenging to find two different inputs that produce the same hash output.

**SHA-1 Algorithm Steps:**

1. **Input Preparation:**
   * Convert the input text into a binary format and append padding bits to ensure the total length is congruent to 448 modulo 512.
   * Append the length of the original message as a 64-bit integer.
2. **Initialization:**
   * Start with predefined hash values (H0, H1, H2, H3, H4) which are 32-bit integers.
3. **Processing Message in Blocks:**
   * The message is divided into 512-bit blocks, and each block is processed through a series of operations that involve bitwise operations, modular additions, and logical functions.
4. **Finalization:**
   * After processing all blocks, the final hash value is obtained by concatenating the hash values, resulting in a 160-bit message digest.

**Applications:**

* **Data Integrity:** Ensures that data has not been altered.
* **Digital Signatures:** Used to verify the authenticity of messages.
* **Password Hashing:** Commonly used to store hashed passwords.

**Limitations:** Despite its popularity, SHA-1 is considered weak against collision attacks, and its use is being phased out in favor of stronger hash functions like SHA-256 and SHA-3.

**Conclusion:** Calculating the SHA-1 message digest is a crucial process in cryptography, providing data integrity and authentication. However, due to its vulnerabilities, it is essential to consider more secure alternatives for sensitive applications.

import hashlib

def main():

    # Input text

    message = input("Enter the message: ")

    # Step 1: Create a SHA-1 hash object

    sha1 = hashlib.sha1()

    # Step 2: Update the hash object with the bytes of the message

    sha1.update(message.encode('utf-8'))

    # Step 3: Calculate the hash digest in hexadecimal format

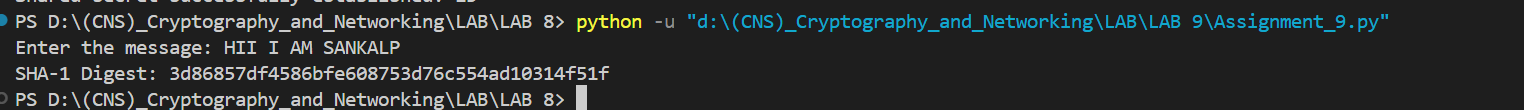
    digest = sha1.hexdigest()

    # Step 4: Output the message digest (in hexadecimal)

    print("SHA-1 Digest:", digest)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

****

**ASSIGNMENT 10**

**Implement the SIGNATURE SCHEME – Digital Signature Standard.**

**Introduction:** The Digital Signature Standard (DSS) is a federal standard for digital signatures that provides a method for ensuring the authenticity and integrity of digital messages or documents. It is widely used in various applications, including secure communications, software distribution, and financial transactions.

**Key Components:**

1. **Digital Signature:** A digital signature is a mathematical scheme for verifying the authenticity and integrity of a message. It serves as a counterpart to handwritten signatures or stamped seals but offers far more inherent security.
2. **Key Pair:** DSS uses asymmetric cryptography, meaning it employs a pair of keys—a private key (known only to the signer) and a public key (shared with recipients).
3. **Hash Function:** DSS utilizes a cryptographic hash function (like SHA-256) to create a fixed-size hash of the message, which is then signed with the private key.

**DSS Process:** The process of creating and verifying a digital signature under DSS involves the following steps:

1. **Key Generation:**
   * A pair of keys (public and private) is generated using a specified algorithm (like DSA - Digital Signature Algorithm).
   * The private key is kept secret, while the public key is distributed to anyone wishing to verify the signature.
2. **Signing a Message:**
   * The sender creates a hash of the original message using a cryptographic hash function.
   * The hash is then signed with the sender's private key to produce the digital signature.
   * The signed hash and the original message are sent to the recipient.
3. **Verifying a Signature:**
   * The recipient receives the message and the digital signature.
   * The recipient generates a hash of the received message using the same hash function.
   * The recipient uses the sender's public key to verify the signature against the generated hash.
   * If the hashes match, it confirms that the message is authentic and has not been altered.

**Applications:**

* **Secure Transactions:** DSS is crucial in e-commerce and financial applications, ensuring the authenticity of transactions.
* **Software Distribution:** Used to verify the integrity and authenticity of software updates or packages.
* **Email Security:** Ensures that emails are not tampered with and verifies the sender’s identity.

**Advantages:**

* **Integrity:** Ensures that the message has not been altered in transit.
* **Authenticity:** Confirms the identity of the sender.
* **Non-repudiation:** The sender cannot deny having sent the message, as only they possess the private key used to sign it.

**Limitations:** DSS relies on the security of the underlying cryptographic algorithms. If vulnerabilities are discovered in the algorithms used for key generation or hashing, the security of the digital signatures can be compromised.

**Conclusion:** The Digital Signature Standard (DSS) is a vital component of modern cryptography, providing a robust mechanism for securing digital communications. By employing a combination of hashing and asymmetric cryptography, DSS ensures the integrity, authenticity, and non-repudiation of digital messages and documents, making it an essential tool in various security-sensitive applications.

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives.asymmetric import dsa

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives import hashes

def generate\_keys():

    # Generate a DSA private key

    private\_key = dsa.generate\_private\_key(key\_size=2048, backend=default\_backend())

    public\_key = private\_key.public\_key()

    # Serialize keys to save them for later use

    private\_pem = private\_key.private\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.TraditionalOpenSSL,

        encryption\_algorithm=serialization.NoEncryption()  # No encryption

    )

    public\_pem = public\_key.public\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    )

    return private\_pem, public\_pem

def sign\_message(private\_key\_pem, message):

    # Load the private key from PEM format (add password=None)

    private\_key = serialization.load\_pem\_private\_key(

        private\_key\_pem,

        password=None,  # Specify None for unencrypted key

        backend=default\_backend()

    )

    # Create a signature for the message

    signature = private\_key.sign(

        message.encode(),

        hashes.SHA256()  # Hashing algorithm

    )

    return signature

def verify\_signature(public\_key\_pem, message, signature):

    # Load the public key from PEM format

    public\_key = serialization.load\_pem\_public\_key(public\_key\_pem, backend=default\_backend())

    # Verify the signature

    try:

        public\_key.verify(

            signature,

            message.encode(),

            hashes.SHA256()  # Hashing algorithm

        )

        return True  # Signature is valid

    except Exception:

        return False  # Signature is invalid

def main():

    # Generate keys

    private\_key, public\_key = generate\_keys()

    # Input message to sign

    message = input("Enter the message to sign: ")

    # Sign the message

    signature = sign\_message(private\_key, message)

    print("Signature:", signature.hex())

    # Verify the signature

    is\_valid = verify\_signature(public\_key, message, signature)

    print("Is the signature valid?", is\_valid)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

