**Final Year B.Tech. (CSE) – VII [ 2024-25]**

**6CS451: Cryptography and Network Security Lab (C&NS Lab)**

**Date: 26/08/2024**

**Assignment 8**

**PRN:** 21510017  **Name:** Onkar Anand Yemul

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

**Ans:**

The Diffie-Hellman Key Exchange is a cryptographic algorithm that allows two parties to securely share a secret key over a public channel. This shared key can then be used for encrypted communication. The algorithm allows two parties to generate a shared secret key that can be used for subsequent encryption and decryption, even if the exchange itself is observed by an eavesdropper.

**How Diffie-Hellman Works:**

1. **Public Parameters:**
   * Both parties agree on a large prime number p and a base g (a primitive root modulo p).
2. **Key Exchange Process:**
   * **Party A** selects a private key ‘a’ and computes A = g^ a mod p, then sends A to Party B.
   * **Party B** selects a private key ‘b’ and computes B = g^ b mod p, then sends B to Party A.
3. **Shared Secret:**
   * **Party A** computes the shared secret as S = B^ a mod p.
   * **Party B** computes the shared secret as S = A^ b mod p.

Since both calculations result in the same value, S becomes the shared secret key, even though an eavesdropper only knows p, g, A, and B.

**The Diffie-Hellman algorithm securely establishes a shared secret key without transmitting it directly, making it fundamental for secure communications in protocols like SSL/TLS.**

To implement the Diffie-Hellman Key Exchange algorithm for client-server communication across two different machines, we will create two Python programs: one for the client and one for the server. The server will generate its public key and share it with the client, and vice versa. Both will then calculate the shared secret key independently.

**Python Code:**

**Client-side program:**

import socket

import random

from sympy import isprime, primerange

def generate\_private\_key(p):

    """Generate a private key."""

    private\_key = random.randint(2, p - 2)

    print(f"\nGenerated Private Key: {private\_key}")

    return private\_key

def calculate\_public\_key(g, private\_key, p):

    """Calculate the public key."""

    public\_key = pow(g, private\_key, p)

    print(f"\nCalculated Public Key: {public\_key}")

    return public\_key

def calculate\_shared\_secret(public\_key, private\_key, p):

    """Calculate the shared secret."""

    shared\_secret = pow(public\_key, private\_key, p)

    return shared\_secret

def find\_next\_prime(n):

    """Find the next prime number greater than or equal to n."""

    if isprime(n):

        return n

    else:

        for prime in primerange(n + 1, n \* 2):

            return prime

def find\_primitive\_roots(p, count=5):

    """Find the first `count` primitive roots of prime `p`."""

    roots = []

    for g in range(2, p):

        seen = set(pow(g, power, p) for power in range(1, p))

        if len(seen) == p - 1:

            roots.append(g)

        if len(roots) >= count:

            break

    return roots

def start\_client(server\_host='localhost', server\_port=5000):

    # Take p as user input and validate it

    p = int(input("\nEnter a prime number (p): "))

    p = find\_next\_prime(p)

    print(f"\nValidated Prime Number (p): {p}")

    # Find and list primitive roots of p

    primitive\_roots = find\_primitive\_roots(p)

    print(f"\nFirst Five Primitive Roots for {p}: {primitive\_roots}")

    # Take g as user input and validate it

    g = int(input(f"\nChoose a primitive root (g) from the above list: "))

    while g not in primitive\_roots:

        print("Invalid choice. Please choose a primitive root from the list.")

        g = int(input(f"\nChoose a primitive root (g) from the above list: "))

    # Generate client's private and public keys

    private\_key = generate\_private\_key(p)

    public\_key = calculate\_public\_key(g, private\_key, p)

    # Create client socket

    client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    client\_socket.connect((server\_host, server\_port))

    # Receive the server's public key

    server\_public\_key = int(client\_socket.recv(1024).decode())

    print(f"\nReceived Server's Public Key: {server\_public\_key}")

    # Send the client's public key to the server

    client\_socket.sendall(str(public\_key).encode())

    # Calculate the shared secret

    shared\_secret = calculate\_shared\_secret(server\_public\_key, private\_key, p)

    print(f"\nShared Secret (Client): {shared\_secret}")

    client\_socket.close()

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

    start\_client()

**Server-side program:**

import socket

import random

from sympy import isprime, primerange

def generate\_private\_key(p):

    """Generate a private key."""

    private\_key = random.randint(2, p - 2)

    print(f"\nGenerated Server's Private Key: {private\_key}")

    return private\_key

def calculate\_public\_key(g, private\_key, p):

    """Calculate the public key."""

    public\_key = pow(g, private\_key, p)

    print(f"\nCalculated Server's Public Key: {public\_key}")

    return public\_key

def calculate\_shared\_secret(public\_key, private\_key, p):

    """Calculate the shared secret."""

    shared\_secret = pow(public\_key, private\_key, p)

    return shared\_secret

def find\_next\_prime(n):

    """Find the next prime number greater than or equal to n."""

    if isprime(n):

        return n

    else:

        for prime in primerange(n + 1, n \* 2):

            return prime

def find\_primitive\_roots(p, count=5):

    """Find the first `count` primitive roots of prime `p`."""

    roots = []

    for g in range(2, p):

        seen = set(pow(g, power, p) for power in range(1, p))

        if len(seen) == p - 1:

            roots.append(g)

        if len(roots) >= count:

            break

    return roots

def start\_server(host='localhost', port=5000):

    # Create server socket

    server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    server\_socket.bind((host, port))

    server\_socket.listen(1)

    print(f"\nServer started. Listening on {host}:{port}")

    # Take p as user input and validate it

    p = int(input("\nEnter a prime number (p): "))

    p = find\_next\_prime(p)

    print(f"\nValidated Prime Number (p): {p}")

    # Find and list primitive roots of p

    primitive\_roots = find\_primitive\_roots(p)

    print(f"\nFirst Five Primitive Roots for {p}: {primitive\_roots}")

    # Take g as user input and validate it

    g = int(input(f"\nChoose a primitive root (g) from the above list: "))

    while g not in primitive\_roots:

        print("Invalid choice. Please choose a primitive root from the list.")

        g = int(input(f"\nChoose a primitive root (g) from the above list: "))

    # Generate server's private and public keys

    private\_key = generate\_private\_key(p)

    public\_key = calculate\_public\_key(g, private\_key, p)

    conn, addr = server\_socket.accept()

    print(f"\nConnected by {addr}")

    # Send the server's public key to the client

    conn.sendall(str(public\_key).encode())

    # Receive the client's public key

    client\_public\_key = int(conn.recv(1024).decode())

    print(f"\nReceived Client's Public Key: {client\_public\_key}")

    # Calculate the shared secret

    shared\_secret = calculate\_shared\_secret(client\_public\_key, private\_key, p)

    print(f"\nShared Secret (Server): {shared\_secret}")

    conn.close()

    server\_socket.close()

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

    start\_server()

# For

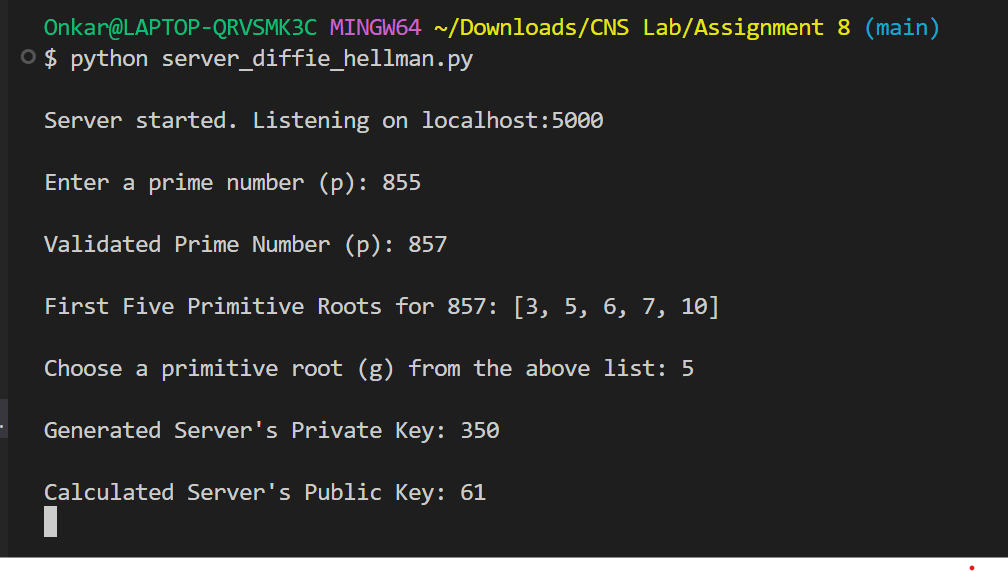
# p: 1014273607262027361

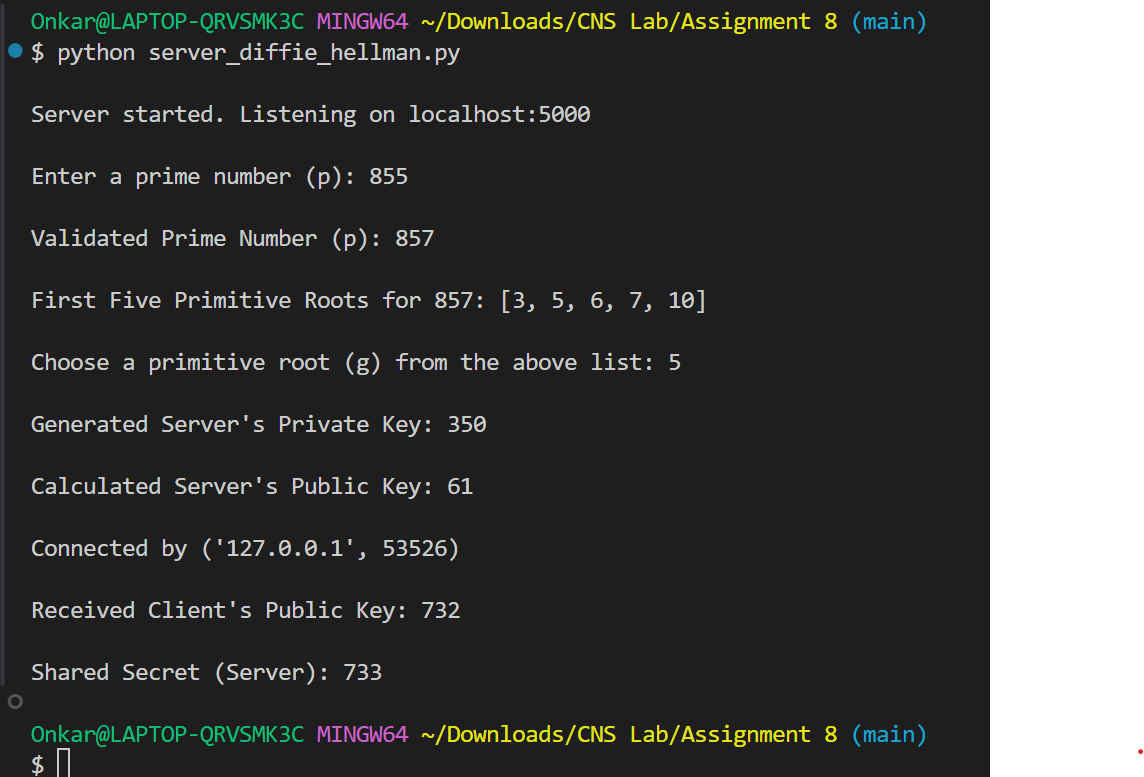
# For

# g: 2

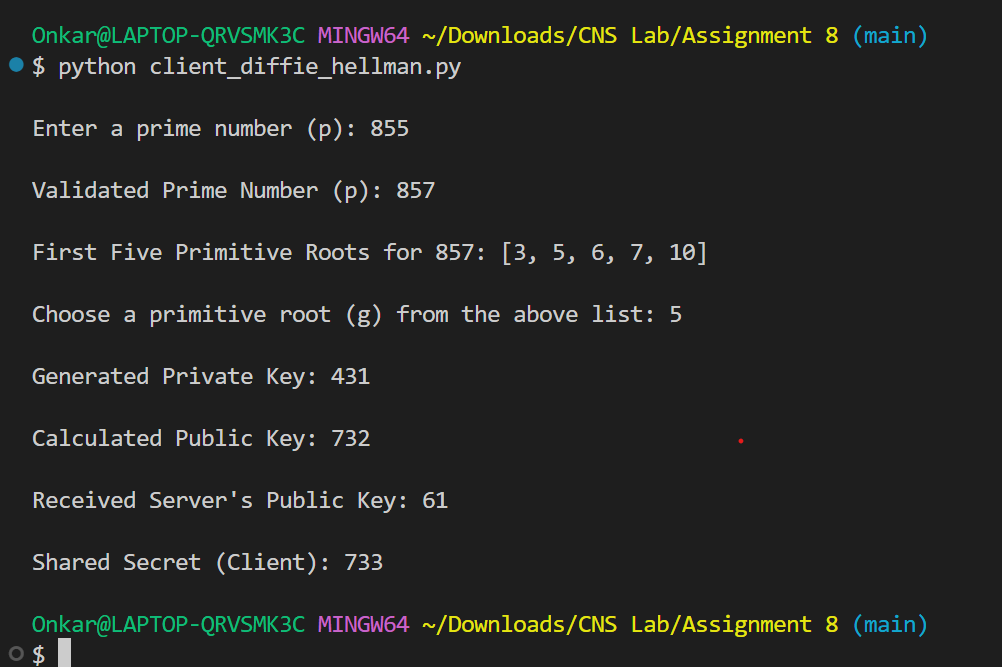
**Output:**

**Server output-**



****

**Client output-**



**Practical Applications of Diffie-Hellman:**

* **Secure Communication:** Establishing a shared secret for symmetric encryption over an insecure channel.
* **VPNs:** Secure key exchange for Virtual Private Networks.
* **TLS/SSL:** Part of the key exchange process in securing internet communications.

The Diffie-Hellman algorithm forms the basis of many modern cryptographic protocols and is crucial for secure communication in distributed systems.