Vivekanand Education Society's Institute of Technology

Department of AI & DS Engineering

Subject: Cryptography and System Security

Class: D11AD

Roll No: 46	**Name: Ashish Patil**
Practical No:3	**Title:Symmetric Key Agreement**
DOP:	**DOS:**
Grades:	**LOs Mapped:**
Signature:	

Title: Symmetric Key Agreement
DOP: 23/1/24
DOS: 2/2/24

Aim: To implement Diffie Hellman Algorithm for symmetric key exchange.
Theory:

1. Steps of the Diffie Hellman Algorithm.
2. Attacks on Diffie Hellman Algorithm.

Steps of the Diffie-Hellman Algorithm:

1. Setup : Both parties, let's call them Alice and Bob, agree on two public parameters:
   - A large prime number \(p \)
   - A primitive root \(g \) modulo \(p \)

2. Private Key Generation :
   - Alice selects a private key \(a \) randomly from the range \(1 < a < p-1 \).
   - Bob selects a private key \(b \) randomly from the range \(1 < b < p-1 \).

3. Public Key Generation :
   - Alice computes her public key \(A \) using the formula \(A = g^a \mod p \).
   - Bob computes his public key \(B \) using the formula \(B = g^b \mod p \).

4. Key Exchange :
   - Alice sends her public key \(A \) to Bob.
   - Bob sends his public key \(B \) to Alice.

5. Shared Secret Calculation :
   - Alice computes the shared secret \(S_A \) using Bob's public key \(B \) and her private key \(a \): \(S_A = B^a \mod p \).
   - Bob computes the shared secret \(S_B \) using Alice's public key \(A \) and his private key \(b \): \(S_B = A^b \mod p \).

6. Shared Secret Agreement :
   - Both Alice and Bob now have a shared secret: \(S_A = S_B \).
   - This shared secret can be used as a symmetric encryption key or for any other secure communication.

Attacks on Diffie-Hellman Algorithm:

1. Man-in-the-Middle (MitM) Attack :
   - In this attack, an attacker intercepts the communication between Alice and Bob and establishes separate key exchanges with each party.
   - The attacker then relays messages between Alice and Bob, decrypting and re-encrypting them using their own keys.
   - To mitigate this attack, Diffie-Hellman key exchange is often combined with digital signatures or other authentication mechanisms to verify the identity of the communicating parties.

2. Discrete Logarithm Problem :
   - The security of the Diffie-Hellman algorithm relies on the assumption that computing discrete logarithms (e.g., finding \(x \) in the equation \(g^x \mod p = y \)) is computationally difficult.
   - With advancements in computational power and algorithms such as the Number Field Sieve, the discrete logarithm problem can be solved for certain parameter choices, potentially compromising the security of Diffie-Hellman key exchange.

3. Small Subgroup Attacks :
   - If the prime modulus \(p \) is poorly chosen or if the generator \(g \) generates a small subgroup of \(Z_p^* \), the security of Diffie-Hellman can be compromised.

- In such cases, an attacker may exploit the mathematical properties of small subgroups to derive the shared secret.
- To mitigate this attack, careful selection of prime modulus \(p \) and generator \(g \) is essential, ensuring they generate a large cyclic group.

4. Pre-computation Attacks :
   - An attacker can pre-compute certain values to accelerate the computation of the shared secret once they obtain the public keys exchanged between Alice and Bob.
   - This can be mitigated by using ephemeral keys, where new private-public key pairs are generated for each key exchange, thus making pre-computation attacks impractical.

Program:


```
import random

def generate_prime():
    # Generate a random large prime number
    while True:
        p = random.randint(100, 1000)
        if is_prime(p):
            return p

def is_prime(n, k=5):
    # Miller-Rabin primality test
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Write n as 2^r * d + 1
    d = n - 1
    r = 0
    while d % 2 == 0:
        d //= 2
        r += 1

    # Witness loop
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
```

```python
            break
        else:
            return False
    return True

def generate_primitive_root(p):
    # Find a primitive root modulo p
    while True:
        g = random.randint(2, p - 1)
        if pow(g, (p - 1) // 2, p) != 1:
            return g

def generate_private_key(p):
    # Generate a random private key
    return random.randint(2, p - 2)

def generate_public_key(g, private_key, p):
    # Generate public key using g^private_key mod p
    return pow(g, private_key, p)

def generate_shared_secret(public_key, private_key, p):
    # Generate shared secret using public_key^private_key mod p
    return pow(public_key, private_key, p)

# Generate large prime number
p = generate_prime()
print("Large prime number (p):", p)
```

```python
# Find primitive root modulo p
g = generate_primitive_root(p)
print("Primitive root (g):", g)

# Generate private keys for Alice and Bob
private_key_alice = generate_private_key(p)
private_key_bob = generate_private_key(p)

# Generate public keys for Alice and Bob
public_key_alice = generate_public_key(g, private_key_alice, p)
public_key_bob = generate_public_key(g, private_key_bob, p)

# Generate shared secret for Alice and Bob
shared_secret_alice = generate_shared_secret(public_key_bob, private_key_alice,
shared_secret_bob = generate_shared_secret(public_key_alice, private_key_bob, p

print("\nAlice's private key:", private_key_alice)
print("Bob's private key:", private_key_bob)

print("\nAlice's public key:", public_key_alice)
print("Bob's public key:", public_key_bob)

print("\nShared secret computed by Alice:", shared_secret_alice)
print("Shared secret computed by Bob:", shared_secret_bob)

# Verify that both shared secrets are equal
assert shared_secret_alice == shared_secret_bob, "Shared secrets do not match!"
```

Output:

```
Large prime number (p): 541
Primitive root (g): 209

Alice's private key: 40
Bob's private key: 139

Alice's public key: 1
Bob's public key: 497

Shared secret computed by Alice: 1
Shared secret computed by Bob: 1
```

Conclusion: We have successfully implemented the diffe-hellman algorithm for symmetric key agreement and studied its vulnerable attacks.p