```python
# Simulating Diffie-Hellman with MITM attack

import random

# Function to simulate power modulo (a^b % p)
def power_mod(base, exp, mod):
    return pow(base, exp, mod)

# Diffie-Hellman participants
class Participant:
    def __init__(self, name, p, g):
        self.name = name
        self.p = p
        self.g = g
        self.private_key = random.randint(2, p - 2)  # Choose private key
        self.public_key = power_mod(g, self.private_key, p)  # Compute public key

    def compute_shared_secret(self, other_public_key):
        return power_mod(other_public_key, self.private_key, self.p)

# Man-in-the-Middle Attacker
class Attacker:
    def __init__(self, p, g):
        self.p = p
        self.g = g
        self.private_key = random.randint(2, p - 2)
        self.public_key = power_mod(g, self.private_key, p)

    def intercept_and_replace(self, original_key):
        # Intercept and return attacker's public key instead of the original
        return self.public_key
```

```
    def compute_shared_secret(self, other_public_key):
        return power_mod(other_public_key, self.private_key, self.p)

# Example of Diffie-Hellman with MITM attack
def main():
    # Prime number (p) and generator (g)
    p = 23  # Small prime number for simplicity
    g = 5   # Generator

    # Participants
    alice = Participant("Alice", p, g)
    bob = Participant("Bob", p, g)
    mallory = Attacker(p, g)  # Man-in-the-middle

    # Step 1: Alice sends her public key to Bob
    alice_public_key = alice.public_key
    intercepted_by_mallory = mallory.intercept_and_replace(alice_public_key)

    # Step 2: Mallory sends her fake public key to Bob
    bob_received_key = intercepted_by_mallory

    # Step 3: Bob sends his public key to Alice
    bob_public_key = bob.public_key
    intercepted_by_mallory_bob = mallory.intercept_and_replace(bob_public_key)

    # Step 4: Mallory sends her fake public key to Alice
    alice_received_key = intercepted_by_mallory_bob

    # Alice computes her "shared secret"
    alice_shared_secret = alice.compute_shared_secret(alice_received_key)
```

```
    # Bob computes his "shared secret"
    bob_shared_secret = bob.compute_shared_secret(bob_received_key)

    # Mallory computes secrets with Alice and Bob
    mallory_shared_with_alice = mallory.compute_shared_secret(alice_public_key)
    mallory_shared_with_bob = mallory.compute_shared_secret(bob_public_key)

    # Outputs
    print("=== Keys ===")
    print(f"Alice's Public Key: {alice_public_key}")
    print(f"Bob's Public Key: {bob_public_key}")
    print(f"Mallory's Fake Public Key: {mallory.public_key}")

    print("\n=== Shared Secrets ===")
    print(f"Alice's Shared Secret: {alice_shared_secret}")
    print(f"Bob's Shared Secret: {bob_shared_secret}")
    print(f"Mallory's Shared Secret with Alice: {mallory_shared_with_alice}")
    print(f"Mallory's Shared Secret with Bob: {mallory_shared_with_bob}")

    print("\n=== Results ===")
    if alice_shared_secret == bob_shared_secret:
        print("Secure Communication: Shared secrets match.")
    else:
        print("Insecure Communication: Man-in-the-middle attack succeeded!")
        print(f"Mallory can decrypt and re-encrypt messages.")

# Run the simulation
main()
```

Here's an explanation of the code, broken down step by step:

Imports

python

Copy code

```python
import random
```

- The random module is used to generate random private keys for the participants and the attacker.

Power Modulo Function

python

Copy code

```python
def power_mod(base, exp, mod):
    return pow(base, exp, mod)
```

- This function calculates (base^exp) % mod efficiently.
- It's used in the Diffie-Hellman process to compute public keys and shared secrets.

Participant Class

python

Copy code

```python
class Participant:
    def __init__(self, name, p, g):
        self.name = name  # Name of the participant (e.g., Alice, Bob)
        self.p = p  # Shared prime number
        self.g = g  # Shared generator
        self.private_key = random.randint(2, p - 2)  # Random private key
        self.public_key = power_mod(g, self.private_key, p)  # Compute public key


    def compute_shared_secret(self, other_public_key):
        return power_mod(other_public_key, self.private_key, self.p)
```

- **Attributes**:

- name: Identifier for the participant.
- p, g: The shared prime number and generator agreed upon by all parties.
- private_key: A randomly generated secret number unique to the participant.
- public_key: Computed as g^{\text{private_key}} \mod p, shared with others.

- **Method**:
  - compute_shared_secret: Computes the shared secret using the other party's public key and their private key.

Attacker Class

python

Copy code

```
class Attacker:
  def __init__(self, p, g):
    self.p = p
    self.g = g
    self.private_key = random.randint(2, p - 2)  # Random private key for attacker
    self.public_key = power_mod(g, self.private_key, p)  # Attacker's public key

  def intercept_and_replace(self, original_key):
    return self.public_key  # Replace intercepted key with attacker's key

  def compute_shared_secret(self, other_public_key):
    return power_mod(other_public_key, self.private_key, self.p)
```

- **Attributes**:
  - Similar to Participant, but the attacker intercepts and manipulates the key exchange.
- **Methods**:
  - intercept_and_replace: Replaces the intercepted key with the attacker's own public key.
  - compute_shared_secret: Computes the shared secret with the participant whose key was intercepted.

Main Function

python

Copy code

```
def main():
    p = 23  # Prime number
    g = 5   # Generator
```

- A small prime number and generator are chosen for simplicity.
- These values are shared by Alice, Bob, and Mallory.

Participants and Attacker Initialization

python

Copy code

```
    alice = Participant("Alice", p, g)
    bob = Participant("Bob", p, g)
    mallory = Attacker(p, g)
```

- **Alice** and **Bob** are initialized as participants, each with their own private and public keys.
- **Mallory** is the attacker with her own private and public keys.

Key Exchange Steps with MITM Attack

1. **Alice Sends Public Key to Bob (Intercepted by Mallory)**:

python

Copy code

```
 alice_public_key = alice.public_key
 intercepted_by_mallory = mallory.intercept_and_replace(alice_public_key)
```

   - Alice’s public key is intercepted by Mallory.
   - Mallory replaces it with her own public key.

2. **Mallory Sends Her Fake Key to Bob**:

python

Copy code

```
 bob_received_key = intercepted_by_mallory
```

   - Bob receives Mallory’s fake key, thinking it is Alice’s.

3. **Bob Sends Public Key to Alice (Intercepted by Mallory)**:

python

Copy code

```
bob_public_key = bob.public_key
intercepted_by_mallory_bob = mallory.intercept_and_replace(bob_public_key)
```

   - Bob’s public key is intercepted and replaced by Mallory’s.

4. **Mallory Sends Her Fake Key to Alice**:

python

Copy code

```
alice_received_key = intercepted_by_mallory_bob
```

   - Alice receives Mallory’s fake key, thinking it is Bob’s.

Compute Shared Secrets

python

Copy code

```
alice_shared_secret = alice.compute_shared_secret(alice_received_key)
bob_shared_secret = bob.compute_shared_secret(bob_received_key)
mallory_shared_with_alice = mallory.compute_shared_secret(alice_public_key)
mallory_shared_with_bob = mallory.compute_shared_secret(bob_public_key)
```

- Alice and Bob compute what they think are shared secrets, but these are different because of Mallory’s interference.
- Mallory computes separate shared secrets with Alice and Bob, allowing her to intercept messages between them.

Display Results

python

Copy code

```
print("=== Keys ===")
print(f"Alice's Public Key: {alice_public_key}")
print(f"Bob's Public Key: {bob_public_key}")
print(f"Mallory's Fake Public Key: {mallory.public_key}")
```

- Displays the public keys involved in the key exchange, including Mallory’s fake key.

python

Copy code

```
print("\n=== Shared Secrets ===")

print(f"Alice's Shared Secret: {alice_shared_secret}")

print(f"Bob's Shared Secret: {bob_shared_secret}")

print(f"Mallory's Shared Secret with Alice: {mallory_shared_with_alice}")

print(f"Mallory's Shared Secret with Bob: {mallory_shared_with_bob}")
```

- Shows the computed shared secrets for all parties.

python

Copy code

```
if alice_shared_secret == bob_shared_secret:

    print("Secure Communication: Shared secrets match.")

else:

    print("Insecure Communication: Man-in-the-middle attack succeeded!")

    print(f"Mallory can decrypt and re-encrypt messages.")
```

- Verifies if the communication is secure or if the MITM attack succeeded.

Key Takeaways

- The code demonstrates how a Man-in-the-Middle (MITM) attack exploits the lack of authentication in the Diffie-Hellman key exchange.
- Authentication mechanisms, such as digital signatures, are essential to prevent this vulnerability.

4o