



### Experiment 8

**Date of Performance:** 17 April 2023

**Date of Submission:** 17 April 2023

**SAP Id:** 60004200132

**Name :** Ayush Jain

**Div:** B

**Batch :** B3

### Aim of Experiment

Implement Diffie Hellman Key exchange protocol. Demonstrate man in middle attack.

### Theory / Algorithm / Conceptual Description:

Diffie-Hellman algorithm

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime  $P$  and  $G$  (a primitive root of  $P$ ) and two private values  $a$  and  $b$ .

$P$  and  $G$  are both publicly available numbers. Users (say Alice and Bob) pick private values  $a$  and  $b$  and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Steps:

Alice	Bob
Public Keys available = $P, G$	Public Keys available = $P, G$
Private Key Selected = $a$	Private Key Selected = $b$
Key generated = $x = G^a \text{ mod } P$	Key generated = $y = G^b \text{ mod } P$
Exchange of generated keys takes place	Exchange of generated keys takes place
Key received = $y$	Key received = $y$

Generated Secret Key = $k_a = y^a \text{ mod } P$	Generated Secret Key = $k_b = x^b \text{ mod } P$
--	--

Algebraically, it can be shown that

$$k_a = k_b$$

Example:

Step 1: Alice and Bob get public numbers  $P = 23$ ,  $G = 9$

Step 2: Alice selected a private key  $a = 4$  and Bob selected a private key  $b = 3$

Step 3: Alice and Bob compute public values Alice:  $x = (9^4 \text{ mod } 23) = (6561 \text{ mod } 23) = 6$  Bob:  $y = (9^3 \text{ mod } 23) = (729 \text{ mod } 23) = 16$

Step 4: Alice and Bob exchange public numbers

Step 5: Alice receives public key  $y = 16$  and Bob receives public key  $x = 6$

Step 6: Alice and Bob compute symmetric keys Alice:  $k_a = y^a \text{ mod } p = 65536 \text{ mod } 23 = 9$  Bob:  $k_b = x^b \text{ mod } p = 216 \text{ mod } 23 = 9$  Step 7: 9 is the shared secret.

## Program

```
from random import randint
from math import sqrt
def isPrime(n):
    if (n <= 1):
        return False
    if (n <= 3):
        return True
    if (n % 2 == 0 or n % 3 == 0):
        return False
    i = 5
    while(i * i <= n):
        if (n % i == 0 or n % (i + 2) == 0) :
            return False
        i = i + 6
    return True
def findPrimefactors(s, n):
    while (n % 2 == 0):
        s.add(2)
        n = n // 2
    for i in range(3, int(sqrt(n)), 2):
        while (n % i == 0):
            s.add(i)
            n = n // i
    if (n > 2):
        s.add(n)
def getPrimitiveRoot(P):
    s = set()
    if (isPrime(P) == False):
```

```
        return -1

    phi = P - 1
    findPrimefactors(s, phi)
    for r in range(2, phi + 1):
        flag = False
        for it in s:
            if (pow(r, phi // it, P) == 1):
                flag = True
                break
        if (flag == False):
            return r
    return -1

P = int(input("Enter value of P: "))
G = getPrimitiveRoot(P)
print('The Value of G is: %d'%(G))
a = randint(2, P-1)
b = randint(2, P-1)
print('The Private Key a for A is: %d'%(a))
print('The Private Key b for B is: %d'%(b))
xa = int(pow(G,a,P))%P
xb = int(pow(G,b,P))%P
ka = int(pow(xb,a,P))%P
kb = int(pow(xa,b,P))%P
print('Secret key for the A is: %d'%(ka))
print('Secret Key for the B is: %d'%(kb))
```

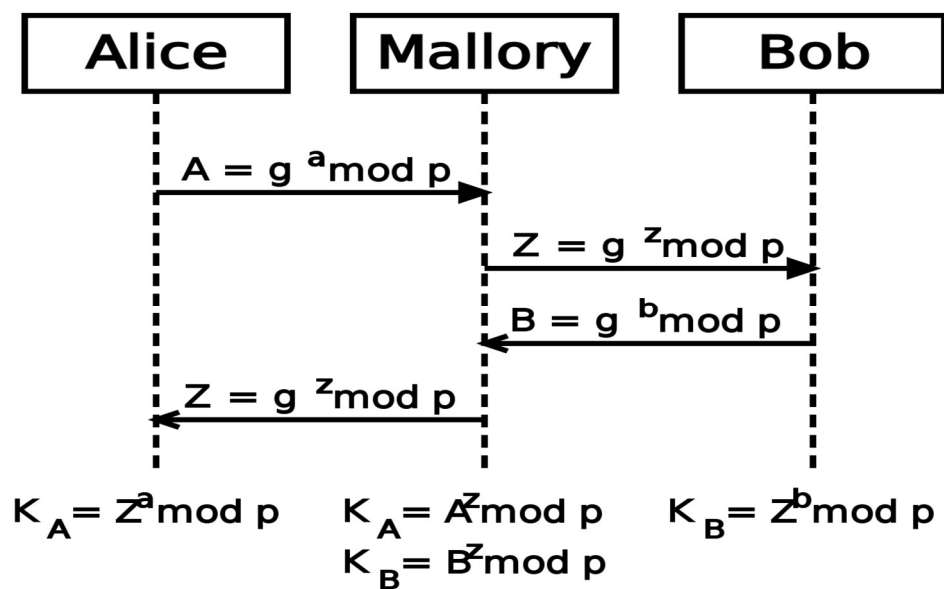
## Output

```
Enter value of P: 47
The Value of G is: 46
The Private Key a for A is: 31
The Private Key b for B is: 25
Secret key for the A is: 46
Secret Key for the B is: 46

...Program finished with exit code 0
Press ENTER to exit console.
```

## Man in the middle attack:

In a man-in-the-middle attack, an attacker intercepts the communication between two parties and impersonates each party to the other. In the context of the Diffie-Hellman key exchange, the attacker would intercept the public keys sent by each party, replace them with their own public key, and then forward them to the intended recipient. The attacker would then compute their own shared secret key with each party, which would be different from the shared secret key that the two parties would compute with each other.



### Step by Step explanation of this process:

Step 1: Selected public numbers  $p$  and  $g$ ,  $p$  is a prime number, called the “modulus” and  $g$  is called the base.

Step 2: Selecting private numbers.

let Alice pick a private random number  $a$  and let Bob pick a private random number  $b$ , Malory picks 2 random numbers  $c$  and  $d$ .

Step 3: Intercepting public values,

Malory intercepts Alice’s public value ( $ga(\text{mod } p)$ ), block it from reaching Bob, and instead sends Bob her own public value ( $gc(\text{mod } p)$ ) and Malory intercepts Bob’s public value ( $gb(\text{mod } p)$ ), block it from reaching Alice, and instead sends Alice her own public value ( $gd(\text{mod } p)$ )

Step 4: Computing secret key

Alice will compute a key  $S1 = gda(\text{mod } p)$ , and Bob will compute a different key,  $S2 = gcb(\text{mod } p)$

Step 5: If Alice uses  $S1$  as a key to encrypt a later message to Bob, Malory can decrypt it, re-encrypt it using  $S2$ , and send it to Bob. Bob and Alice won’t notice any problem and may assume their communication is encrypted, but in reality, Malory can decrypt, read, modify, and then re-encrypt all their conversations.

To **prevent** man-in-the-middle attacks in the Diffie-Hellman key exchange, the two parties should authenticate each other's public keys using digital signatures or a trusted third party. Additionally, the communication channel should be encrypted using a secure encryption algorithm, such as AES, to prevent eavesdropping by the attacker.

### Code:

```
import random

# public keys are taken
# p is a prime number
# g is a primitive root of p
p = int(input('Enter a prime number: '))
g = int(input('Enter a primitive root of p: '))

class Alice:

    def __init__(self):
        # Generating a random private number selected by Alice
```

```
self.private_key = random.randint(1, p)

def publish(self):
    # Generating public value
    return pow(g, self.private_key, p)

def compute_secret_key(self, public_key):
    # Computing secret key
    return pow(public_key, self.private_key, p)

class Bob:
    def __init__(self):
        # Generating a random private number selected for Bob
        self.private_key = random.randint(1, p)

    def publish(self):
        # Generating public value
        return pow(g, self.private_key, p)

    def compute_secret_key(self, public_key):
        # Computing secret key
        return pow(public_key, self.private_key, p)

class Mallory:
    def __init__(self):
        # Generating a random private number selected for Mallory
        self.private_key_a = random.randint(1, p)
        self.private_key_b = random.randint(1, p)
```

```
def publish(self, i):
    # Generating public value
    if i == 0:
        return pow(g, self.private_key_a, p)
    else:
        return pow(g, self.private_key_b, p)
def compute_secret_key(self, public_key, i):
    # Computing secret key
    if i == 0:
        return pow(public_key, self.private_key_a, p)
    else:
        return pow(public_key, self.private_key_b, p)
alice = Alice()
bob = Bob()
mallory = Mallory()
# Printing out the private selected number by Alice, Bob, and Mallory
print(f"Alice selected (a): {alice.private_key}")
print(f"Bob selected (b): {bob.private_key}")
print(f"Mallory selected private number for Alice (c): {mallory.private_key_a}")
print(f"Mallory selected private number for Bob (d): {mallory.private_key_b}")
# Generating public values
ga = alice.publish()
gb = bob.publish()
gca = mallory.publish(0)
gcb = mallory.publish(1)
print(f"Alice published (ga): {ga}")
print(f"Bob published (gb): {gb}")
print(f"Mallory published value for Alice (gc): {gca}")
```



```
print(f'Mallory published value for Bob (gd): {gcb}')
```

# Computing the secret key


```
sa = alice.compute_secret_key(gca)
sb = bob.compute_secret_key(gcb)
sma = mallory.compute_secret_key(ga, 0)
smb = mallory.compute_secret_key(gb, 1)
print(f'Alice computed (S1): {sa}')
```

print(f'Mallory computed key for Alice (S1): {sma}')

print(f'Bob computed (S2): {sb}')

print(f'Mallory computed key for Bob (S2): {smb}')

## Output:



```
Enter a prime number: 227
Enter a primitive root of p: 14
Alice selected (a): 93
Bob selected (b): 46
Mallory selected private number for Alice (c): 175
Mallory selected private number for Bob (d): 140
Alice published (ga): 123
Bob published (gb): 222
Mallory published value for Alice (gc): 32
Mallory published value for Bob (gd): 16
Alice computed (S1): 20
Mallory computed key for Alice (S1): 20
Bob computed (S2): 81
Mallory computed key for Bob (S2): 81

...Program finished with exit code 0
Press ENTER to exit console.█
```

The use of a fixed generator  $g$  could potentially lead to a small subgroup attack if the attacker can choose the generator. Additionally, the code does not perform any authentication, which means that a man-in-the-middle (MITM) attack is possible, where Mallory intercepts the public keys and forwards them to Alice and Bob while impersonating each other, allowing Mallory to eavesdrop on their communication.

To fix these issues, the implementation should use a secure random generator for  $g$ , and perform authentication to prevent MITM attacks.

**Conclusion:** We successfully implemented Diffie Hellman Key exchange protocol. Demonstrate man in middle attack.