Coursework 1 - Exercise 4

November 9, 2023

a) Describe the traditional Diffie-Hellman key exchange protocol.

b) Design a protocol based on the traditional Diffie-Hellman key exchange, that allows 3 parties P_1 ,P_2 and P_3 to exchange a single symmetric key K .The following conditions have to be fulfilled:

- Only the parties P_1 ,P_2 and P_3 can know the key K.
- The key should be verified by all parties.

You can give your solution as a sequence of message sent from P_i to P_j . e.g. $P_i \xrightarrow{\text{m}} P_j$. We assume a prime p and the generator g of the cyclic group $\mathbb{Z}_p^*$ to be publicly known.

---

a)

The Diffie-Hellman key exchange protocol is based on asymmetric cryptography, so it will be envolving public and private keys.

1. The two envolved parties trying to communicate(Alice & Bob) will agree on two public values: p and g
2. Both Alice and Bob choose a random private key (a, b) limited by a specific size(e.g. 128 bits)
3. Alice and Bob calculate their public keys:

$$A = g^a \mod p, B = g^b \mod p$$

4. Alice and Bob share their public key
5. The shared secret can now be calculated like:

$$k = B^a \mod p = A^b \mod p$$

6. If Alice wants to send a message m to Bob, Alice would encrypt the message using the shared secret(key)

$$c = E(m, k)$$

7. and Bob will decrypt it using the same key

$$m = D(c, k)$$

b)

To achieve a secure key exchange among three parties, P_1, P_2, and P_3, while ensuring that only these parties know the key and that the key is verified by all, we can design a protocol based on the traditional Diffie-Hellman key exchange with some additional steps. Here's a step-by-step sequence of messages sent among the parties:

1. **Setup**: Public parameters are known to all parties: a large prime number "p" and a generator "g" of the cyclic group $\mathbb{Z}_p^*$.

2. **Key Generation**:
   - Each party generates their private key:
     - P_1: Chooses a random private key a_1.
     - P_2: Chooses a random private key a_2.
     - P_3: Chooses a random private key a_3.
   - Each party calculates their public key:
     - $A_1 = g^{a_1} \mod p$
     - $A_2 = g^{a_2} \mod p$
     - $A_3 = g^{a_3} \mod p$

3. **Public Key Exchange**:
   - Parties exchange their public keys with one other user:
     - $P_1 \xrightarrow{A_1} P_2$
     - $P_2 \xrightarrow{A_2} P_3$
     - $P_3 \xrightarrow{A_3} P_1$

4. **Public *"Secret"* Calculation**
   - Each party generates an intermediate Public/Secret key:
     - $A_{31} = A_3^{a_1} \mod p$
     - $A_{12} = A_1^{a_2} \mod p$
     - $A_{23} = A_2^{a_3} \mod p$

5. **Public *"Secret"* Exchange**
   - Parties exchange their public keys with one other user:
     - $P_1 \xrightarrow{A_{31}} P_2$
     - $P_2 \xrightarrow{A_{12}} P_3$
     - $P_3 \xrightarrow{A_{23}} P_1$

6. **Shared Secret Calculation**:
   - Each party calculates a shared secret key with others using their private keys and received public keys:
     - P_1 computes $K_{231} = A_{23}^{a_1} \mod p$
     - P_2 computes $K_{312} = A_{31}^{a_2} \mod p$
     - P_3 computes $K_{123} = A_{12}^{a_3} \mod p$

7. Key Verification:
   - To ensure key verification, each party shares a cryptographic hash of the calculated shared secrets:
     - $P_1 \xrightarrow{H_1 = H(K_{231})} P_2, P_3$
     - $P_2 \xrightarrow{H_2 = H(K_{312})} P_1, P_3$

- $P_3 \xrightarrow{H_3=H(K_{123})} P_1, P_2$

8. Verification Check:

- Each party verifies the received hash values:
  - P_1 checks if H_2 matches the received hash from P_2 and H_3 matches the received hash from P_3.
  - P_2 checks if H_1 matches the received hash from P_1 and H_3 matches the received hash from P_3.
  - P_3 checks if H_1 matches the received hash from P_1 and H_2 matches the received hash from P_2.

9. Final Symmetric Key Derivation:

If all parties successfully verify the received hash values, they can trust that they all share the same secret.

And then we should see how the condition:

$$H_1 = H_2 = H_3$$

And also:

$$K_{231} = K_{312} = K_{123} = (g^{a_1 \cdot a_2 \cdot a_3} \mod p)$$

We can see the implementation in python:

[1]:
```
g, p, [a1, a2, a3] = 7, 11, [6, 9, 8]
A1, A2, A3 = pow(g, a1, p), pow(g, a2, p), pow(g, a3, p)
X1, X2, X3 = pow(A2, a1, p), pow(A1, a2, p), pow(A1, a3, p)
X1, X2, X3 = pow(X1, a3, p), pow(X2, a3, p), pow(X3, a2, p)
print("X1 =", X1)
print("X2 =", X2)
print("X3 =", X3)
print("Z =", pow(g, a1*a2*a3, p))

# Veryfing the keys with hash function
import hashlib
hash1 = hashlib.sha256()
hash1.update(str(X1).encode('utf-8'))
hash2 = hashlib.sha256()
hash2.update(str(X2).encode('utf-8'))
hash3 = hashlib.sha256()
hash3.update(str(X3).encode('utf-8'))
h1 = hash1.hexdigest()
h2 = hash2.hexdigest()
h3 = hash3.hexdigest()
assert h1 == h2 == h3
print("\nHash of X1, X2, X3 =", h1)
```

```
X1 = 5
X2 = 5
X3 = 5
Z = 5

Hash of X1, X2, X3 =
ef2d127de37b942baad06145e54b0c619a1f22327b2ebbcfbec78f5564afe39d
```

If we analize the succession of operations, we can see that for two users the secret key was calculated like:

$$((g^{a_1} \mod p)^{a_2} \mod p) = g^{a_1 \cdot a_2} \mod p$$

And for three

$$\left(((g^{a_1} \mod p)^{a_2} \mod p)^{a_3} \mod p\right) = g^{a_1 \cdot a_2 \cdot a_3} \mod p$$

So we can generalize the secret key exchage like:

[2]:
```
import random
import numpy as np

n = 5 # number of participants
key_size = 64 # key size in bits
g, p = 17, 19 # public parameters

# generate random private keys
a = [random.randint(0, 2^key_size) for _ in range(n)]

# compute public keys
A = [g] * n
for i in range(n):
    A = [pow(A[j], a[i], p) for j in range(n)]



# compute shared secret
s = pow(g, int(np.prod(a)), p)

print(A, s)

# check if all shared secrets are equal
assert all(num == s for num in A), "Shared secrets are not equal"
print("Success")

# verify hashes, this step is not necessary since we already checked if all␣
↪shared secrets are equal
hashes = [hashlib.sha256() for _ in range(n)]
for i in range(n):
    hashes[i].update(str(A[i]).encode('utf-8'))
    for j in range(n):
```

```
        hashes[i].update(str(A[j]).encode('utf-8'))
    assert hashes[i].hexdigest() == hashes[0].hexdigest(), "Hashes are not␣
 ↪equal"

print("Hashes are equal:", hashes[0].hexdigest())
```

```
[16, 16, 16, 16, 16] 16
Success
Hashes are equal:
de1df27a9383b7354117450177b839dc1c1c7d2885876fd23f32dcadcb43258d
```

This method allows a single secret key for all participants where we obtain intermediate non-secret keys between all the subset of users, requiring all participants to communicate with all other participants and allowing the private keys to stay private while also keeping the secrey key from anyone who didn't participate.