ALEKHYA THOGITI

ASSIGNMENT-17

1.Explain Data Encryption Standard (DES) and Rivest-Shamir-Adleman (RSA) Algorithms.

Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of electronic data. It was developed in the 1970s and was adopted as a federal standard in the United States in 1977. DES was once widely used in various applications, including securing ATM transactions and electronic communications.

Features:

1. **Symmetric Key Encryption:** DES uses the same key for both encryption and decryption, meaning both the sender and receiver must have the same key.
2. **Block Cipher:** DES operates on fixed-size blocks of data, specifically 64-bit blocks.
3. **Key Size:** DES uses a 56-bit key for encryption, though the effective key length is technically 64 bits with 8 bits used for parity.
4. **Rounds:** The algorithm performs 16 rounds of processing on the data. Each round involves a series of permutations and substitutions based on the key.
5. **Feistel Structure:** DES uses a Feistel network structure, where the data block is split into two halves, and the process is applied iteratively.

Strengths and Weaknesses:

Strengths: DES was considered highly secure when it was first introduced and played a key role in advancing the field of cryptography.

Weaknesses: The 56-bit key size is now considered too small, making DES vulnerable to brute-force attacks. Advances in computing power have rendered DES insecure for many applications.

Successor:

Due to its vulnerabilities, DES has largely been replaced by the Advanced Encryption Standard (AES), which offers stronger security with larger key sizes and more robust encryption techniques.

Rivest-Shamir-Adleman (RSA)

RSA is an asymmetric cryptographic algorithm used for secure data transmission. It was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman, and is widely used for secure data transmission, digital signatures, and key exchange.

Features:

1. **Asymmetric Key Encryption:** RSA uses a pair of keys: a public key for encryption and a private key for decryption. The public key can be shared openly, while the private key is kept secret.
2. **Mathematical Foundation:** RSA's security is based on the mathematical difficulty of factoring large composite numbers into their prime factors.
3. **Key Size:** RSA keys are typically 1024, 2048, or 4096 bits long. Larger key sizes provide greater security but require more computational resources.
4. **Encryption and Decryption:** The encryption process involves raising the plaintext to a power (the public key) and then taking the modulus of a product of two large primes (the modulus). Decryption involves raising the ciphertext to a power (the private key) and then taking the modulus of the same product.

Process:

Key Generation: Generate two large prime numbers, multiply them to get the modulus, and derive the public and private keys from these values.

Encryption: Encrypt a message by raising it to the power of the public key and taking the modulus.

Decryption: Decrypt the message by raising the ciphertext to the power of the private key and taking the modulus.

Strengths and Weaknesses:

Strengths: RSA provides robust security and is widely used in securing internet communications, including SSL/TLS for web browsers, email encryption, and digital signatures.

Weaknesses: RSA is computationally intensive, especially with larger key sizes. It is slower compared to symmetric-key algorithms like AES for encrypting large amounts of data.

Applications:

RSA is commonly used for secure key exchange, where a symmetric key can be securely shared between parties using RSA encryption. It is also used for digital signatures, where the authenticity of a message can be verified.

DES: A symmetric-key block cipher with a 56-bit key, considered obsolete due to vulnerability to brute-force attacks.

RSA: An asymmetric-key algorithm used for secure data transmission and digital signatures, based on the difficulty of factoring large numbers.

2. Explain Diffie-Hellman Key Exchange Algorithm With an Example

Diffie-Hellman Key Exchange Algorithm

The Diffie-Hellman Key Exchange algorithm is a method for securely exchanging cryptographic keys over a public channel. It was one of the first practical implementations of public key exchange and is used to establish a shared secret between two parties, which can then be used for encrypting subsequent communications with a symmetric key algorithm.

Key Features:

Asymmetric Key Exchange: Involves the use of public and private keys but not for encryption and decryption. Instead, it establishes a shared secret key.

Mathematical Foundation: Based on the difficulty of solving the discrete logarithm problem in finite fields.

Security: Relies on the computational difficulty of determining the shared secret from the public values.

Steps:

1. Both parties agree on two large prime numbers, p (a prime number) and g (a primitive root modulo p).
2. Each party generates a private key, a and b, which are random numbers kept secret.
3. Each party computes a public value based on their private key
- Party 1 (Alice) computes A=gamod pA = g^a \mod pA=gamodp
- Party 2 (Bob) computes B=gbmod pB = g^b \mod pB=gbmodp
4. The public values A and B are exchanged over the public channel.
5. Each party then computes the shared secret key using the other party's public value and their own private key:
- Alice computes the shared secret as s=Bamod ps = B^a \mod ps=Bamodp
- Bob computes the shared secret as s=Abmod ps = A^b \mod ps=Abmodp
6. Both Alice and Bob now have the same shared secret key, s, which can be used for secure communication.

Example:

Step-by-Step Example:

1. **Agree on public values:**

   - Prime number p=23p = 23p=23
   - Primitive root g=5g = 5g=5
2. **Private keys:**
   - Alice's private key a=6a = 6a=6
   - Bob's private key b=15b = 15b=15
3. **Compute public values:**
   - Alice computes A=gamod p=56mod 23=15625mod 23=8A = g^a \mod p = 5^6 \mod 23 = 15625 \mod 23 = 8A=gamodp=56mod23=15625mod23=8
   - Bob computes B=gbmod p=515mod 23=30517578125mod 23=19B = g^b \mod p = 5^{15} \mod 23 = 30517578125 \mod 23 = 19B=gbmodp=515mod23=30517578125mod23=19
4. **Exchange public values:**
   - Alice sends A=8A = 8A=8 to Bob
   - Bob sends B=19B = 19B=19 to Alice
5. **Compute shared secret:**
   - Alice computes s=Bamod p=196mod 23=47045881mod 23=2s = B^a \mod p = 19^6 \mod 23 = 47045881 \mod 23 = 2s=Bamodp=196mod23=47045881mod23=2
   - Bob computes s=Abmod p=815mod 23=35184372088832mod 23=2s = A^b \mod p = 8^{15} \mod 23 = 35184372088832 \mod 23 = 2s=Abmodp=815mod23=35184372088832mod23=2
6. **Result:**
   - Both Alice and Bob now share the secret key s=2s = 2s=2.

Security Considerations:

- The security of the Diffie-Hellman Key Exchange relies on the difficulty of computing the discrete logarithm, which is considered computationally infeasible for large prime numbers.
- The values of ppp and ggg should be chosen carefully to ensure security.
- Diffie-Hellman is vulnerable to man-in-the-middle attacks if not authenticated, meaning an attacker could intercept and modify the public values exchanged between the parties. To prevent this, Diffie-Hellman can be combined with digital signatures or other authentication methods.

3. Explain Digital Signature Algorithm (DSA) With an Example.

Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) is a Federal Information Processing Standard for digital signatures. It was proposed by the National Institute of Standards and Technology (NIST) in 1991 for use in their Digital Signature Standard (DSS). DSA is used to generate a digital signature, which can be used to verify the authenticity and integrity of a message.

Key Features:

1. **Asymmetric Key Algorithm:** DSA uses a pair of keys, a private key for signing and a public key for verification.
2. **Mathematical Foundation:** Based on the mathematical principles of modular exponentiation and discrete logarithms.
3. **Security:** The security of DSA is based on the difficulty of solving the discrete logarithm problem.

Components:

- **p:** A large prime number, typically between 512 and 1024 bits.
- **q:** A 160-bit prime factor of p−1p-1p−1.
- **g:** A number less than ppp such that g=h(p−1)/qmod pg = h^{(p-1)/q} \mod pg=h(p−1)/qmodp where hhh is any number less than ppp and h(p−1)/qmod p>1h^{(p-1)/q} \mod p > 1h(p−1)/qmodp>1.
- **Private Key (x):** A randomly selected integer such that 0<x<q0 < x < q0<x<q.
- **Public Key (y):** Calculated as y=gxmod py = g^x \mod py=gxmodp.

Steps for Signing and Verifying a Message

1. **Key Generation:**
   - Choose parameters ppp, qqq, and ggg.
   - Select a private key xxx randomly from the interval (0,q)(0, q)(0,q).
   - Compute the public key y=gxmod py = g^x \mod py=gxmodp.
2. **Signing a Message:**
   - Generate a random integer kkk such that 0<k<q0 < k < q0<k<q.
   - Compute r=(gkmod p)mod qr = (g^k \mod p) \mod qr=(gkmodp)modq.
   - Compute the hash of the message H(m)H(m)H(m).
   - Compute s=(k−1(H(m)+x⋅r))mod qs = (k^{-1} (H(m) + x \cdot r)) \mod qs=(k−1(H(m)+x⋅r))modq.

- The signature of the message is the pair (r,s)(r, s)(r,s).

3. **Verifying a Signature:**
    - Verify that 0<r<q0 < r < q0<r<q and 0<s<q0 < s < q0<s<q; if not, the signature is invalid.
    - Compute the hash of the message H(m)H(m)H(m).
    - Compute w=s−1mod qw = s^{-1} \mod qw=s−1modq.
    - Compute u1=(H(m)⋅w)mod qu_1 = (H(m) \cdot w) \mod qu1=(H(m)⋅w)modq and u2=(r⋅w)mod qu_2 = (r \cdot w) \mod qu2=(r⋅w)modq.
    - Compute v=((gu1⋅yu2)mod p)mod qv = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod qv=((gu1⋅yu2)modp)modq.
    - The signature is valid if and only if v=rv = rv=r.

Example:

1. **Key Generation:**
    - Let p=23p = 23p=23, q=11q = 11q=11, and g=2g = 2g=2.
    - Choose private key x=6x = 6x=6.
    - Compute public key y=gxmod p=26mod 23=64mod 23=18y = g^x \mod p = 2^6 \mod 23 = 64 \mod 23 = 18y=gxmodp=26mod23=64mod23=18.
2. **Signing a Message:**
    - Let the message be "Hello".
    - Compute the hash H(m)H(m)H(m). For simplicity, let's assume H(m)=9H(m) = 9H(m)=9.
    - Choose random k=3k = 3k=3.
    - Compute r=(gkmod p)mod q=(23mod 23)mod 11=8mod 11=8r = (g^k \mod p) \mod q = (2^3 \mod 23) \mod 11 = 8 \mod 11 = 8r=(gkmodp)modq=(23mod23)mod11=8mod11=8.
    - Compute s=(k−1(H(m)+x⋅r))mod qs = (k^{-1} (H(m) + x \cdot r)) \mod qs=(k−1(H(m)+x⋅r))modq.
        - Compute k−1mod qk^{-1} \mod qk−1modq: 3−1mod 11=43^{-1} \mod 11 = 43−1mod11=4 (since 3×4mod 11=13 \times 4 \mod 11 = 13×4mod11=1).
        - Compute s=(4×(9+6⋅8))mod 11=(4×(9+48))mod 11=(4×57)mod 11=228mod 11 =8s = (4 \times (9 + 6 \cdot 8)) \mod 11 = (4 \times (9 + 48)) \mod 11

= (4 \times 57) \mod 11 = 228 \mod 11 = 8s=(4×(9+6·8))mod11=(4×(9+48))mod11=(4×57)mod11=228mod11=8.

- The signature is (r,s)=(8,8)(r, s) = (8, 8)(r,s)=(8,8).

3. **Verifying a Signature:**
   - Verify 0<r<q0 < r < q0<r<q and 0<s<q0 < s < q0<s<q: Both conditions are true.
   - Compute the hash H(m)=9H(m) = 9H(m)=9.
   - Compute w=s−1mod q=8−1mod 11=7w = s^{-1} \mod q = 8^{-1} \mod 11 = 7w=s−1modq=8−1mod11=7 (since 8×7mod 11=18 \times 7 \mod 11 = 18×7mod11=1).
   - Compute u1=(H(m)·w)mod q=(9×7)mod 11=63mod 11=8u_1 = (H(m) \cdot w) \mod q = (9 \times 7) \mod 11 = 63 \mod 11 = 8u1 =(H(m)·w)modq=(9×7)mod11=63mod11=8.
   - Compute u2=(r·w)mod q=(8×7)mod 11=56mod 11=1u_2 = (r \cdot w) \mod q = (8 \times 7) \mod 11 = 56 \mod 11 = 1u2 =(r·w)modq=(8×7)mod11=56mod11=1.
   - Compute v=((gu1·yu2)mod p)mod q=((28·181)mod 23)mod 11=(256·18mod 23)mod 11=(4608mod 23)mod 11=(2)mod 11=8v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q = ((2^8 \cdot 18^1) \mod 23) \mod 11 = (256 \cdot 18 \mod 23) \mod 11 = (4608 \mod 23) \mod 11 = (2) \mod 11 = 8v=((gu1·yu2)modp)modq=((28·181)mod23)mod11=(256·18mod23)mod11=(4608mod23) mod11=(2)mod11=8.
   - Since v=rv = rv=r, the signature is valid.

DSA is a widely used digital signature algorithm that ensures the authenticity and integrity of a message. It relies on the difficulty of the discrete logarithm problem and involves key generation, message signing, and signature verification processes.

4. Explain the Following Types of One-time Password (OTP) Algorithms with Examples: a. Time-based OTP (TOTP) b. HMAC-based OTP (HOTP)

One-time Password (OTP) Algorithms

One-time passwords (OTPs) are passwords that are valid for only one login session or transaction, enhancing security by reducing the risk of password interception or replay attacks. Two common OTP algorithms are Time-based OTP (TOTP) and HMAC-based OTP (HOTP).

a. Time-based OTP (TOTP)

Overview: TOTP is an extension of HOTP and generates OTPs based on the current time. This means the generated OTP is valid only for a short time period, typically 30 seconds. TOTP is widely used in two-factor authentication (2FA) systems.

Key Features:

- **Time-dependent:** The OTP is generated based on the current time and a shared secret key.
- **Short lifespan:** OTPs are typically valid for a short duration (e.g., 30 seconds), increasing security.
- **Synchronization:** Both the server and client must have synchronized clocks.

Algorithm:

1. **Shared Secret (K):** A shared secret key between the client and server.
2. **Time Step (T):** The length of time each OTP is valid, usually 30 seconds.
3. **Current Time (C):** The current Unix time divided by the time step, which creates a time counter.

Steps:

1. Calculate the current time counter TC=⌊Current Unix Time/T⌋T_C = \lfloor \text{Current Unix Time} / T \rfloorTC=⌊Current Unix Time/T⌋.
2. Compute the HMAC hash using the shared secret key and the time counter.
3. Extract a truncated value from the HMAC hash.
4. Generate the OTP by taking the truncated value modulo 10d10^d10d, where ddd is the desired number of digits.

Example:

1. **Shared Secret (K):** JBSWY3DPEHPK3PXP
2. **Time Step (T):** 30 seconds
3. **Current Time (C):** Assume Unix time is 1627286400 (2021-07-26 00:00:00 UTC)
4. Calculate the time counter: TC=⌊1627286400/30⌋=54242880T_C = \lfloor 1627286400 / 30 \rfloor = 54242880TC=⌊1627286400/30⌋=54242880

Generating OTP:

- Compute HMAC-SHA1 hash: HMAC-SHA1(K, T_C)
- Assume hash result (in hexadecimal): 0x1f8698690e02ca16618550ef7f19da8e945b555a

- Extract dynamic truncation value: The last nibble of the hash gives an offset (0xA), extracting 4 bytes starting at offset 10: 0x50ef7f19
- Convert to integer and modulo 10610^6106: 0x50ef7f19 (decimal 1357871129) mod 10610^6106 = 871129

Thus, the OTP is 871129.

b. HMAC-based OTP (HOTP)

Overview: HOTP generates OTPs based on a counter value. Each OTP is valid until used, making it suitable for transaction-based authentication systems. The OTP changes only when the counter increments.

Key Features:

- **Counter-based:** The OTP is generated using a counter value and a shared secret key.
- **Event-driven:** The OTP changes only when the counter is incremented, allowing flexibility in time.

Algorithm:

1. **Shared Secret (K):** A shared secret key between the client and server.
2. **Counter (C):** An incremental counter value.

Steps:

1. Compute the HMAC hash using the shared secret key and the counter value.
2. Extract a truncated value from the HMAC hash.
3. Generate the OTP by taking the truncated value modulo 10d10^d10d, where ddd is the desired number of digits.

Example:

1. **Shared Secret (K):** JBSWY3DPEHPK3PXP
2. **Counter (C):** 1

Generating OTP:

- Compute HMAC-SHA1 hash: HMAC-SHA1(K, C)
- Assume hash result (in hexadecimal): 0x4fd3b597b4a0f4dbd7d4cc6304c7aee34ed8b90c
- Extract dynamic truncation value: The last nibble of the hash gives an offset (0xC), extracting 4 bytes starting at offset 12: 0x4cc6304c
- Convert to integer and modulo 10610^6106: 0x4cc6304c (decimal 1284756652) mod 10610^6106 = 756652

Thus, the OTP is 756652.

- **TOTP:** Generates time-based OTPs, providing high security by ensuring the OTP is valid only for a short time period, requiring synchronized clocks between the client and server.
- **HOTP:** Generates counter-based OTPs, suitable for event-driven scenarios where the OTP remains valid until used, without needing synchronized time between client and server.