CYBER SECURITY

ASSIGNMENT -17

NAME:B.Shanmukh

Reg.no: 282023-024

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

ANS:

Data Encryption Standard (DES)

Overview:

DES is a symmetric-key algorithm for the encryption of digital data. Developed in the early 1970s and adopted as a federal standard in 1977 by the National Institute of Standards and Technology (NIST), DES was widely used for secure data transmission.

Key Characteristics:

- **Symmetric Key Algorithm**: Both the sender and receiver use the same key for encryption and decryption.
- **Block Cipher**: DES encrypts data in fixed-size blocks of 64 bits.
- **Key Length**: Uses a 56-bit key for encryption, although the input key is 64 bits long, 8 bits are used for parity checking.
- **Feistel Structure**: DES uses a Feistel network, which divides the block into two halves and processes them through multiple rounds of permutation and substitution.

How DES Works:

1. **Initial Permutation (IP)**: The 64-bit block undergoes an initial permutation.
2. **Rounds**: The block is then processed through 16 rounds of Feistel operations. Each round involves:
    - Splitting the block into left (L) and right (R) halves.
    - Applying a round-specific function (F) to the right half and then XORing it with the left half.
    - Swapping the halves.
3. **Final Permutation (FP)**: After 16 rounds, the final permutation is applied to the combined block.

Security:

- DES has been considered insecure due to its short key length (56 bits), which makes it vulnerable to brute-force attacks.
- It has been replaced by the Advanced Encryption Standard (AES) in many applications.

Rivest-Shamir-Adleman (RSA)

Overview:

RSA is an asymmetric-key algorithm used for secure data transmission. It was developed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. Unlike DES, RSA relies on a pair of keys: a public key for encryption and a private key for decryption.

Key Characteristics:

- **Asymmetric Key Algorithm**: Uses a pair of keys – a public key for encryption and a private key for decryption.
- **Based on Number Theory**: RSA's security relies on the mathematical difficulty of factoring large composite numbers.
- **Variable Key Length**: Typical key lengths are 1024, 2048, or 4096 bits, with longer keys offering higher security.

How *RSA* Works:

1. **Key Generation**:
   - Select two large prime numbers, ppp and qqq.
   - Compute n=pqn = pqn=pq. nnn is used as the modulus for both the public and private keys.
   - Compute the totient ϕ(n)=(p−1)(q−1)\phi(n) = (p-1)(q-1)ϕ(n)=(p−1)(q−1).
   - Choose an integer eee (public exponent) such that 1<e<ϕ(n)1 < e < \phi(n)1<e<ϕ(n) and eee is coprime with ϕ(n)\phi(n)ϕ(n).
   - Compute ddd (private exponent) such that d≡e−1mod ϕ(n)d \equiv e^{-1} \mod \phi(n)d≡e−1modϕ(n).
2. **Public and Private Keys**:
   - Public key: (e,n)(e, n)(e,n).
   - Private key: (d,n)(d, n)(d,n).
3. **Encryption**:
   - To encrypt a message MMM, convert MMM to an integer mmm such that 0≤m<n0 \leq m < n0≤m<n.
   - Compute the ciphertext ccc as c≡memod nc \equiv m^e \mod nc≡memodn.
4. **Decryption**:
   - To decrypt the ciphertext ccc, compute the original message mmm as m≡cdmod nm \equiv c^d \mod nm≡cdmodn.
   - Convert mmm back to the original message MMM.

Security:

- The security of RSA is based on the computational difficulty of factoring large composite numbers.
- As computing power increases, the recommended key lengths also increase to ensure security.

Comparison:

- **Key Type**: DES uses symmetric keys, while RSA uses asymmetric keys.
- **Key Length**: DES has a fixed key length of 56 bits, whereas RSA uses variable key lengths (commonly 1024, 2048, or 4096 bits).
- **Use Cases**: DES is typically used for bulk data encryption, while RSA is often used for secure key exchange, digital signatures, and encryption of small data blocks.
- **Security**: DES is vulnerable to brute-force attacks due to its short key length, while RSA's security depends on the difficulty of factoring large numbers, making it more secure with appropriate key sizes.

In summary, DES and RSA serve different purposes in the realm of cryptography, with DES being a fast, symmetric-key algorithm for general data encryption, and RSA being a robust, asymmetric-key algorithm used primarily for secure key exchange and authentication.

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

ANS:

The Diffie-Hellman Key Exchange algorithm is a method used to securely exchange cryptographic keys over a public channel. It allows two parties to generate a shared secret key, which can then be used for encrypted communication, even if the initial exchange occurs in the presence of potential eavesdroppers.

Overview of Diffie-Hellman Key Exchange

Key Concepts:

1. **Public Parameters**:
   - A large prime number ppp (also known as the modulus).
   - A primitive root ggg of ppp (also known as the base or generator).
2. **Private Keys**:
   - Each party generates their own private key, which is a secret number chosen randomly.
3. **Public Keys**:
   - Each party computes their public key using the formula gprivate keymod pg^{\text{private key}} \mod pgprivate keymodp.
4. **Shared Secret**:
   - Both parties use the other's public key and their own private key to compute the shared secret.

Steps in Diffie-Hellman Key Exchange:

1. **Agreement on Public Parameters**:
   - Alice and Bob agree on a large prime number ppp and a base ggg.
2. **Private Key Selection**:
   - Alice chooses a private key aaa (a random number).
   - Bob chooses a private key bbb (a random number).
3. **Public Key Computation**:
   - Alice computes her public key AAA using A=gamod pA = g^a \mod pA=gamodp.
   - Bob computes his public key BBB using B=gbmod pB = g^b \mod pB=gbmodp.
4. **Exchange of Public Keys**:
   - Alice sends her public key AAA to Bob.
   - Bob sends his public key BBB to Alice.
5. **Shared Secret Computation**:
   - Alice computes the shared secret sss using s=Bamod ps = B^a \mod ps=Bamodp.
   - Bob computes the shared secret sss using s=Abmod ps = A^b \mod ps=Abmodp.

Because B=gbmod pB = g^b \mod pB=gbmodp and A=gamod pA = g^a \mod pA=gamodp, both parties will end up with the same shared secret s=gabmod ps = g^{ab} \mod ps=gabmodp.

Example of Diffie-Hellman Key Exchange:
Step-by-Step Example:

1. **Agreement on Public Parameters**:
   - Alice and Bob agree on a prime number p=23p = 23p=23 and a base g=5g = 5g=5.
2. **Private Key Selection**:
   - Alice chooses her private key a=6a = 6a=6.
   - Bob chooses his private key b=15b = 15b=15.
3. **Public Key Computation**:
   - Alice computes her public key: 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 his public key: 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 of Public Keys**:
   - Alice sends her public key A=8A = 8A=8 to Bob.
   - Bob sends his public key B=19B = 19B=19 to Alice.
5. **Shared Secret Computation**:
   - Alice computes the shared secret: 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 the shared secret: s=Abmod p=815mod 23=35184372088832mod 23=2s = A^b \mod p = 8^{15} \mod 23 = 35184372088832 \mod 23 = 2s=Abmodp=815mod23=35184372088832mod23=2.

Both Alice and Bob have computed the same shared secret s=2s = 2s=2. This shared secret can now be used as a key for symmetric encryption to securely communicate further.

Security Considerations:

- The security of the Diffie-Hellman Key Exchange relies on the difficulty of computing discrete logarithms, which means that given gamod pg^a \mod pgamodp and ppp, it is computationally infeasible to determine aaa.
- The choice of a large prime ppp and a suitable base ggg is critical to ensure the security of the key exchange process.

In summary, the Diffie-Hellman Key Exchange allows two parties to securely generate a shared secret key over an insecure channel, forming the basis for secure communication.

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

ANS:

The Digital Signature Algorithm (DSA) is a Federal Information Processing Standard for digital signatures. Digital signatures are used to verify the authenticity and integrity of a message, software, or digital document. DSA is based on the mathematical concept of modular exponentiation and discrete logarithms.

Key Concepts of DSA:

1. **Key Generation**:
   - **Prime Number ppp**: A large prime number.
   - **Subprime qqq**: A prime divisor of p−1p-1p−1.
   - **Base ggg**: A number such that g=h(p−1)/qmod pg = h^{(p-1)/q} \mod pg=h(p−1)/qmodp, where hhh is any integer with 1<h<p−11 < h < p-11<h<p−1 and h(p−1)/qmod p>1h^{(p-1)/q} \mod p > 1h(p−1)/qmodp>1.
2. **Private Key**:
   - A randomly chosen integer xxx such that 0<x<q0 < x < q0<x<q.
3. **Public Key**:
   - Computed as y=gxmod py = g^x \mod py=gxmodp.

Digital Signature Creation:

1. **Hashing**:
   - The message MMM to be signed is hashed using a cryptographic hash function (e.g., SHA-1), producing a hash value H(M)H(M)H(M).

2. **Signature Generation**:
    - Choose 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 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, where k−1k^{-1}k−1 is the modular inverse of kmod qk \mod qkmodq.
3. **Signature**:
    - The signature of the message is the pair (r,s)(r, s)(r,s).

Digital Signature Verification:

1. **Hashing**:
    - The verifier hashes the received message MMM to get the hash value H(M)H(M)H(M).
2. **Verification**:
    - 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.
    - Compute 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.
3. **Validation**:
    - The signature is valid if and only if v=rv = rv=r.

Example of DSA:

Key Generation:

1. **Public Parameters**:
    - Choose p=23p = 23p=23, q=11q = 11q=11, and g=4g = 4g=4.
2. **Private Key**:
    - Choose x=6x = 6x=6.
3. **Public Key**:
    - Compute y=gxmod p=46mod 23=4096mod 23=18y = g^x \mod p = 4^6 \mod 23 = 4096 \mod 23 = 18y=gxmodp=46mod23=4096mod23=18.

Signature Creation:

1. **Message Hashing**:
    - Let the message MMM be "hello".
    - Assume the hash H(M)=9H(M) = 9H(M)=9 (for simplicity).
2. **Signature Generation**:
    - Choose k=3k = 3k=3.
    - Compute r=(gkmod p)mod q=(43mod 23)mod 11=64mod 23mod 11=9r = (g^k \mod p) \mod q = (4^3 \mod 23) \mod 11 = 64 \mod 23 \mod 11 = 9r=(gkmodp)modq=(43mod23)mod11=64mod23mod11=9.
    - 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 (the modular inverse of 3 modulo 11), which is 4.
        - Compute s=4⋅(9+6⋅9)mod 11=4⋅(9+54)mod 11=4⋅63mod 11=252mod 11=10s = 4 \cdot (9 + 6 \cdot 9) \mod 11 = 4 \cdot (9 + 54) \mod 11 = 4 \cdot 63 \mod 11 = 252 \mod 11 = 10s=4⋅(9+6⋅9)mod11=4⋅(9+54)mod11=4⋅63mod11=252mod11=10.
3. **Signature**:
    - The signature is (r,s)=(9,10)(r, s) = (9, 10)(r,s)=(9,10).

Signature Verification:

1. **Message Hashing**:
   - Hash the received message "hello" to get H(M)=9H(M) = 9H(M)=9.
2. **Verification**:
   - Compute w=s−1mod qw = s^{-1} \mod qw=s−1modq, the modular inverse of 10 modulo 11, which is 10.
   - Compute u1=(H(M)·w)mod q=(9·10)mod 11=90mod 11=2u_1 = (H(M) \cdot w) \mod q = (9 \cdot 10) \mod 11 = 90 \mod 11 = 2u1=(H(M)·w)modq=(9·10)mod11=90mod11=2.
   - Compute u2=(r·w)mod q=(9·10)mod 11=90mod 11=2u_2 = (r \cdot w) \mod q = (9 \cdot 10) \mod 11 = 90 \mod 11 = 2u2=(r·w)modq=(9·10)mod11=90mod11=2.
   - Compute v=((gu1·yu2)mod p)mod q=((42·182)mod 23)mod 11=(16·324)mod 23mod 11=5184mod 23mod 11=18mod 11=7v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q = ((4^2 \cdot 18^2) \mod 23) \mod 11 = (16 \cdot 324) \mod 23 \mod 11 = 5184 \mod 23 \mod 11 = 18 \mod 11 = 7v=((gu1·yu2)modp)modq=((42·182)mod23)mod11=(16·324)mod23mod11=5184mod23mod11=18mod11=7.
3. **Validation**:
   - The signature is valid if v=rv = rv=r. In this case, v=7v = 7v=7 and r=9r = 9r=9, so the signature is invalid.

In this example, the signature validation failed due to the parameters chosen for simplicity and illustration. In practice, large primes and secure hash functions are used to ensure the validity and security of DSA signatures.

4. Explain the Following Types of One-time Password (OTP) Algorithms with Examples:

a. Time-based OTP (TOTP)

b. HMAC-based OTP (HOTP)

ANS:

One-time Password (OTP) algorithms are used to generate a password that is valid for only one authentication session or transaction. Here are explanations and examples for Time-based OTP (TOTP) and HMAC-based OTP (HOTP) algorithms:

a.Time-based OTP (TOTP)

Overview:

TOTP is a type of OTP that uses the current time as a source of uniqueness. It combines a shared secret key with the current timestamp to generate a unique, time-based OTP. This method ensures that the OTP changes at regular intervals, typically every 30 or 60 seconds.

How TOTP Works:

1. **Shared Secret**: Both the client (user device) and the server share a secret key.
2. **Current Time**: The current time is divided into fixed intervals (time steps).
3. **Generation**:
   - The current time is divided by the time step (e.g., 30 seconds) to get the time counter.
   - This time counter is concatenated with the secret key.
   - The concatenated string is hashed using a cryptographic hash function (e.g., HMAC-SHA1).
   - The hash is then truncated to produce a fixed-length OTP (usually 6-8 digits).

Example:

1. **Parameters**:
   - Shared secret key: `JBSWY3DPEHPK3PXP`
   - Time step: 30 seconds
   - Current time: `2024-05-30T12:00:00Z` (UTC)
2. **Time Counter Calculation**:
   - Convert the current time to Unix time (number of seconds since 1970-01-01T00:00:00Z): `1717113600`
   - Divide by the time step: `1717113600 / 30 = 57237120`
3. **Concatenate and Hash**:
   - Use HMAC-SHA1 to hash the time counter concatenated with the secret key.
   - HMAC-SHA1(`JBSWY3DPEHPK3PXP`, `57237120`)
4. **Truncate**:
   - Extract a 6-digit OTP from the hash value.

   Suppose the resulting OTP is `123456`.

Use Case:

- A user wants to log in to a website that supports TOTP. They enter the OTP generated by an authenticator app (like Google Authenticator) that uses the shared secret key and the current time to produce the OTP.

b. HMAC-based OTP (HOTP)

Overview:

HOTP is an OTP algorithm that generates a password based on a counter value, which increments with each OTP request. Unlike TOTP, HOTP does not rely on the current time, making it event-based rather than time-based.

How HOTP Works:

1. **Shared Secret**: Both the client (user device) and the server share a secret key.
2. **Counter**: The counter value increments with each OTP generation.
3. **Generation**:
   - The counter is concatenated with the secret key.
   - The concatenated string is hashed using a cryptographic hash function (e.g., HMAC-SHA1).
   - The hash is then truncated to produce a fixed-length OTP (usually 6-8 digits).

Example:

1. **Parameters**:
   - Shared secret key: `JBSWY3DPEHPK3PXP`
   - Counter value: `1`
2. **Concatenate and Hash**:
   - Use HMAC-SHA1 to hash the counter value concatenated with the secret key.
   - HMAC-SHA1(`JBSWY3DPEHPK3PXP`, `1`)
3. **Truncate**:
   - Extract a 6-digit OTP from the hash value.

   Suppose the resulting OTP is `654321`.

4. **Increment Counter**:
   - For the next OTP generation, increment the counter to `2`.

Use Case:

- A user requests an OTP to authenticate a transaction. The system generates an OTP using the shared secret key and the current counter value, then increments the counter. The user enters the OTP to complete the transaction.

Comparison:

- **Dependency**: TOTP depends on the current time, whereas HOTP depends on a counter.
- **Use Case**: TOTP is typically used for time-sensitive authentication (e.g., logging in), while HOTP is suitable for event-based scenarios (e.g., token-based authentication).
- **Synchronization**: TOTP requires time synchronization between the server and the client. HOTP requires synchronization of the counter, which can lead to desynchronization if not managed properly.

In summary, both TOTP and HOTP are widely used OTP algorithms, each suited to different scenarios based on their dependency on time or event counters.