Unit II

Fundamentals of Algorithms in Cybersecurity

Assignment Questions

Q1. Explain Data Encryption Standard (DES) and Rivest-Shamir-Adleman (RSA) Algorithms.

Data Encryption Standard (DES)

DES is a symmetric-key algorithm for the encryption of digital data.

Developed by IBM in the 1970s and adopted as a standard by the National Institute of Standards and Technology (NIST) in 1977.

It was widely used for securing sensitive but unclassified material.

Key Characteristics:

Symmetric Key Encryption: Uses the same key for both encryption and decryption.

Block Cipher: Encrypts data in fixed-size blocks (64 bits for DES).

Key Size: 56-bit key, which is relatively small by modern standards, leading to vulnerabilities.

Structure: Based on a Feistel network, which divides the block into two halves and processes them through multiple rounds of permutation and substitution.

Rounds: Consists of 16 rounds of processing.

How It Works:

Initial Permutation (IP): The plaintext is permuted.

Rounds: Each round includes:

Expansion Permutation (E): Expands the right half from 32 to 48 bits.

XOR with Key: The expanded right half is XORed with the round key.

Substitution (S-Boxes): The 48-bit result is divided into 6-bit chunks, each fed into a substitution box (S-box), producing a 32-bit output.

Permutation (P): The 32-bit result is permuted.

XOR: The permuted output is XORed with the left half from the previous round.

Final Permutation (FP): The halves are swapped and combined, followed by another permutation.

Security:

DES is no longer considered secure due to its short key length, making it susceptible to brute-force attacks.

It has been succeeded by more secure algorithms like AES (Advanced Encryption Standard).

Rivest-Shamir-Adleman (RSA)

RSA is an asymmetric cryptographic algorithm used for secure data transmission.

Developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977.

Based on the mathematical difficulty of factoring large integers.

Key Characteristics:

Asymmetric Key Encryption: Uses a pair of keys - a public key for encryption and a private key for decryption.

Public-Key Cryptography: The public key can be shared openly, while the private key is kept secret.

Key Size: Commonly ranges from 1024 to 4096 bits, providing varying levels of security.

How It Works:

Key Generation:Two large prime numbers (p and q) are chosen.

Compute

$$n = p \ q$$

The security of RSA relies on the computational difficulty of factoring the product of two large prime numbers.

As computing power increases, longer keys are required to ensure security. Currently, 2048-bit keys are common, with 3072-bit or longer keys recommended for future-proofing.

Applications:

RSA is used in various security protocols, such as SSL/TLS for secure web communications, digital signatures, and encryption of sensitive data.

Q2. Explain Diffie-Hellman Key Exchange Algorithm With an Example

Diffie-Hellman Key Exchange Algorithm

- The Diffie-Hellman Key Exchange algorithm is a method for two parties to securely share a secret key over an insecure communication channel.

- It was introduced by Whitfield Diffie and Martin Hellman in 1976.

- This algorithm allows two parties to establish a shared secret that can be used for encrypted communication, without having to share the secret key beforehand.

Key Characteristics:

Public Key Exchange:Both parties exchange information publicly but can compute the shared secret key privately.

-Security Basis: The security relies on the difficulty of solving the discrete logarithm problem in a large prime number field.

How It Works:

1. Public Parameters:

   - A large prime number p.

   - A primitive root modulo p (also known as a generator) g.

2. Private Keys:

   - Each party selects a private key:

     - Alice chooses a private key a (a random integer).

     - Bob chooses a private key b (a random integer).

3. Public Keys:

   - Each party computes their public key:

     - Alice computes $A = g^a \mod p$ and sends it to Bob.

     - Bob computes $B = g^b \mod p$ and sends it to Alice.

4. Shared Secret:

   - Each party uses their private key and the other party’s public key to compute the shared secret:

- Alice computes the shared secret $s = B^a \mod p$.

- Bob computes the shared secret $s = A^b \mod p$.

Since $B = g^b \mod p$ and $A = g^a \mod p$, the shared secret s will be the same for both Alice and Bob:

$$s = (g^b \mod p)^a \mod p = g^{ba} \mod p$$

$$s = (g^a \mod p)^b \mod p = g^{ab} \mod p$$

Example

Let's go through a simplified example using small numbers:

1. Public Parameters

- Prime number $p = 23$.

- Primitive root $g = 5$.

2. Private Keys:

- Alice selects $a = 6$.

- Bob selects $b = 15$.

3. Public Keys:

- Alice computes $A = g^a \mod p = 5^6 \mod 23 = 15625 \mod 23 = 8$.

- Bob computes $B = g^b \mod p = 5^{15} \mod 23 = 30517578125 \mod 23 = 19$.

Alice sends $A = 8$ to Bob, and Bob sends $B = 19$ to Alice.

4. Shared Secret:

- Alice computes the shared secret $s = B^a \mod p = 19^6 \mod 23 = 47045881 \mod 23 =$.

- Bob computes the shared secret $s = A^b \mod p = 8^{15} \mod 23 = 35184372088832 \mod 23 = 2$.

Therefore, both Alice and Bob now share the same secret $s = 2$.

Security

The security of the Diffie-Hellman Key Exchange lies in the computational difficulty of the discrete logarithm problem. Even if an attacker knows the public values p, g, A, and B, it is computationally infeasible to derive the private keys a or b and thus the shared secret s. This makes the Diffie-Hellman Key Exchange a robust method for securely exchanging cryptographic keys over an insecure channel.

Q3. 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 and adopted as FIPS-186 in 1993.

- DSA is based on the mathematical principles of modular exponentiation and discrete logarithms.

Key Characteristics:

- Asymmetric Cryptography: Uses a pair of keys - a private key for signing and a public key for verifying signatures.

- Security Basis: The security relies on the difficulty of computing discrete logarithms.

- Digital Signature: Ensures the authenticity and integrity of a message.

How It Works:

1. Key Generation:

   - Choose a large prime number p.

   - Choose a 160-bit prime number q such that q divides $p-1$.

   - Choose a number g where $g = h^{(p-1)/q} \mod p$ and h is any integer such that $1 < h < p-1$.

   - Choose a private key x such that $0 < x < q$.

   - Compute the public key y where $y = g^x \mod p$.

2. Signature Generation:

- Select a random integer k such that $0 < k < q$.

- Compute $r = (g^k \mod p) \mod q$.

- Compute $s = (k^{-1} \cdot (H(m) + x \cdot r)) \mod q$ where $H(m)$ is the hash of the message m.

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

3. Signature Verification:

- Verify that $0 < r < q$ and $0 < s < q$.

- Compute $w = s^{-1} \mod q$.

- Compute $u_1 = (H(m) \cdot w) \mod q$ and $u_2 = (r \cdot w) \mod q$.

- Compute $v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q$.

- The signature is valid if $v = r$.

Example

Let's go through a simplified example using small numbers to illustrate the process:

1. Key Generation:

- Choose $p = 23$.

- Choose $q = 11$ (note that in practice, p and q are much larger).

- Choose $g = 4$ (such that $g = h^{(p-1)/q} \mod p$).

- Choose a private key $x = 3$.

- Compute the public key $y = g^x \mod p = 4^3 \mod 23 = 64 \mod 23 = 18$.

2. Signature Generation:

- Select a random $k = 6$.

- Compute $r = (g^k \mod p) \mod q = (4^6 \mod 23) \mod 11 = 4096 \mod 23 = 15 \mod 11 = 4$.

- Assume $H(m) = 5$ (hash of the message).

- Compute $s = (k^{-1} \cdot (H(m) + x \cdot r)) \mod q$.

  - $k^{-1} \mod q = 6^{-1} \mod 11 = 2$ (since $6 \cdot 2 = 12 \equiv 1 \mod 11$).

  - $s = (2 \cdot (5 + 3 \cdot 4)) \mod 11 = (2 \cdot (5 + 12)) \mod 11 = (2 \cdot 17) \mod 11 = 34 \mod 11 = 1$.

- The signature is $(r, s) = (4, 1)$.

3. Signature Verification:

- Verify that $0 < r < q$ and $0 < s < q$, which is true for $r = 4$ and $s = 1$.

- Compute $w = s^{-1} \mod q = 1^{-1} \mod 11 = 1$.

- Compute $u_1 = (H(m) \cdot w) \mod q = (5 \cdot 1) \mod 11 = 5$.

- Compute $u_2 = (r \cdot w) \mod q = (4 \cdot 1) \mod 11 = 4$.

- Compute $v = ((g^{u_1} \cdot y^{u_2}) \mod p) \mod q = ((4^5 \cdot 18^4) \mod 23) \mod 11$.

  - $4^5 = 1024 \mod 23 = 12$.

  - $18^4 = 104976 \mod 23 = 9$.

  - $v = (12 \cdot 9 \mod 23) \mod 11 = 108 \mod 23 = 16 \mod 11 = 5$.

Since $v \neq r$, the signature would be invalid in this example. However, this simplified illustration demonstrates the steps involved in DSA. In a real-world scenario with appropriately large values, v should equal r for a valid signature.

Q4. Explain the Following Types of One-time Password (OTP) Algorithms with Examples:

a. Time-based OTP (TOTP)

One-time Password (OTP) Algorithms

One-time passwords (OTPs) are dynamic, temporary codes used to authenticate users for a single session or transaction. They provide an extra layer of security by being valid only for a short period or a single use. Two popular OTP algorithms are Time-based OTP (TOTP) and HMAC-based OTP (HOTP).

- TOTP is an extension of HOTP that includes a time factor to ensure the OTP is only valid for a short duration.

- It is defined in RFC 6238.

- TOTP relies on a shared secret key and the current time to generate the OTP.

How It Works:

1. Shared Secret: Both the server and the client have a pre-shared secret key.

2. Time Interval: The current time is divided into fixed intervals (e.g., 30 seconds).

3. Hash Function: The current time interval is hashed along with the secret key using HMAC (Hash-based Message Authentication Code).

4. OTP Generation: The output is truncated to generate a 6 to 8-digit OTP.

Example:

Let's illustrate TOTP with an example:

1. Parameters:

   - Shared Secret Key: `JBSWY3DPEHPK3PXP` (Base32 encoded)

   - Current Time: Let's assume the current Unix time is `1617818463` (seconds since epoch).

2. Time Interval:

   - Interval: 30 seconds

   - Current Time Step: $\text{Time Step} = \left\lfloor \frac{\text{Current Unix Time}}{30} \right\rfloor = \left\lfloor \frac{1617818463}{30} \right\rfloor = 53927282$

3. HMAC Calculation:

   - Convert the time step to an 8-byte array: `0000000003367C82`

   - Compute HMAC-SHA1 of the time step using the secret key.

4. Truncate and Generate OTP:

   - Extract a 4-byte dynamic binary code from the HMAC result.

   - Convert the binary code to an integer.

   - Compute the OTP: $\text{OTP} = (\text{Dynamic Binary Code}) \mod 10^6$.

For simplicity, let's assume the HMAC-SHA1 result yields an integer that, when truncated, gives `123456`. Therefore, the OTP is `123456`.

b. HMAC-based OTP (HOTP)

- HOTP is a counter-based OTP algorithm defined in RFC 4226.

- It generates OTPs based on a counter value and a shared secret key.

- The counter ensures each OTP is unique for each authentication event.

How It Works:

1. Shared Secret: Both the server and the client share a secret key.

2. Counter:A counter value is incremented with each OTP generation.

3. Hash Function: The counter value is hashed with the secret key using HMAC.

4. OTP Generation:The output is truncated to generate a 6 to 8-digit OTP.

Example:

Let's illustrate HOTP with an example:

1. Parameters:

   - Shared Secret Key: `JBSWY3DPEHPK3PXP` (Base32 encoded)

   - Counter Value: Assume the counter value is `1`.

2. HMAC Calculation:

   - Convert the counter value to an 8-byte array: `0000000000000001`.

   - Compute HMAC-SHA1 of the counter value using the secret key.

3. Truncate and Generate OTP:

   - Extract a 4-byte dynamic binary code from the HMAC result.

   - Convert the binary code to an integer.

   - Compute the OTP: $\text{OTP} = (\text{Dynamic Binary Code}) \mod 10^6$.

For simplicity, let's assume the HMAC-SHA1 result yields an integer that, when truncated, gives `654321`. Therefore, the OTP is `654321`.

Summary

-TOTP generates OTPs based on the current time and a shared secret, ensuring the OTP is valid only for a short, fixed duration.

- HOTP generates OTPs based on a counter value and a shared secret, ensuring each OTP is unique and used once.