Question 1

Diffie-Hellman (DH) Key Exchange: Bob's Public Key Size and MITM Attack Discussion

(a) Maximum Number of Bits in Bob's Public Key (h_1)

In the Diffie-Hellman key exchange protocol, each party generates a public key based on the following formula:

$$h = g^x \bmod q$$

where:

- g is the generator of the group,
- q is a large prime modulus (the size of the group),
- x is the party's private key (a randomly chosen integer),
- h is the resulting public key.

In the given question:

- The size of the prime modulus |q| is 2048 bits.

Since h is computed as $g^x \ mod \ q$, the output h is an element of the group Z_q, which means h is a number between 0 and $q-1$. Thus, the size (in bits) of Bob's public key h_1 will be at most the size of q itself.

- In other words, h_1 will have at most 2048 bits.
- The public key cannot be larger than the modulus q, because it is reduced modulus q.
- Therefore, Bob's public key h_1 can have at most 2048 bits.

(b) Is the Man-in-the-Middle (MITM) Attack on DH an Eavesdropping Attack?

Short Answer:

No, a MITM attack on Diffie-Hellman is not just an eavesdropping attack; it is an active attack.

An eavesdropping attack means that an adversary passively listens to the communication between two parties without altering the messages.

In contrast, in a Man-in-the-Middle (MITM) attack on Diffie-Hellman:

- The attacker **intercepts** and **modifies** the exchanged public keys.
- The attacker **pretends** to be Bob when communicating with Alice, and **pretends** to be Alice when communicating with Bob.
- Specifically:
    1. Alice sends her public key to Bob — but the attacker intercepts it and sends their own forged public key to Bob instead.
    2. Bob sends his public key to Alice — but again, the attacker intercepts it and sends a forged key to Alice.
- As a result:
    - Alice thinks she shares a secret with Bob (but actually shares it with the attacker).
    - Bob thinks he shares a secret with Alice (but actually shares it with the attacker).

Thus, the attacker establishes two separate shared keys:

- One with Alice,
- One with Bob.

Using these two keys, the attacker can:

- Decrypt the messages,
- Read/modify them if desired,
- Re-encrypt and forward them to the intended receiver without either party knowing.

Question 2

Deterministic Encryption vs Randomized Encryption in Eavesdropping Indistinguishability Experiment

(a) How the Adversary Guarantees 100% Success with Deterministic Encryption

Background:

In the **eavesdropping indistinguishability experiment** (from slide 14, Week-14), the following basic steps occur:

1. The challenger generates a key pair (pk, sk).
2. The adversary submits two plaintexts m_0 and m_1.
3. The challenger randomly selects a bit $b \in \{0,1\}$, and encrypts m_b using the public key to generate ciphertext c.
4. The adversary is then given c and must guess which message was encrypted (i.e., guess b).

The goal is that the adversary should not be able to distinguish whether m_0 or m_1 was encrypted.

When Encryption is Deterministic (c := Enc(pk, m_b)):

- **Deterministic encryption** means:
  - Same message always produces the same ciphertext.
  - There is no randomness involved in the encryption process.
- Thus, if the adversary knows the public key pk, they can precompute:
  - $c_0 = \text{Enc}(pk, m_0)$
  - $c_1 = \text{Enc}(pk, m_1)$
- Now, when the challenger sends the ciphertext ccc, the adversary can simply:
  - Compare c with c_0 and c_1.
  - If $c = c_0$, then $b = 0$.
  - If $c = c_1$, then $b = 1$.
- Therefore, the adversary can always correctly guess b with 100% success.

(b) Why the Adversary Fails if Encryption is Randomized (c ← Enc(pk, m_b))

When Encryption is Randomized:

- In randomized encryption, each time you encrypt a message, the encryption algorithm introduces randomness.
- As a result:
  - Same plaintext produces different ciphertexts each time.
  - Ciphertexts are unpredictable even if the plaintext is the same.
- In this case:
  - The adversary cannot precompute c_0 and c_1 ahead of time.
  - Even if the adversary encrypts m_0 and m_1 themselves to get c_0 and c_1, the ciphertext produced by the challenger will look different because of the randomness.
- Thus, the adversary has no advantage over random guessing.
  - The best they can do is guess b with probability 50% (random guess).

Question 3

Breaking El Gamal Encryption if Discrete Logarithms Become Easy (PPT)

Background:

The security depends on the difficulty of the discrete logarithm problem (DLP). The discrete logarithm problem states: Given g, q, and $h=g^x \bmod q$, it is hard to compute x.. El Gamal assumes that without knowing the private key xxx, an adversary cannot decrypt the ciphertext. However, if discrete logarithms could be inverted in polynomial time (PPT), this assumption would no longer hold.

El Gamal Encryption Scheme (Quick Review):

Key Generation:

- Choose a large prime q and generator g of Z^*_q (multiplicative group modulo q).
- Choose a random private key $x \in Z^*_q$.
- Compute the public key:

$$h=g^x \bmod q$$

- Public key: (q, g, h)
- Private key: x

Encryption:

- To encrypt a message $m \in Z^*_q$
  - Choose a random $y \in Z^*_q$
  - Compute:

$$c_1 = g^y \bmod q$$
$$c_2 = m \times h^y \bmod q$$

- The ciphertext is the pair (c_1, c_2).

Decryption:

- To decrypt (c_1, c_2):
  - Compute:

$$s = c_1^x \mod q$$

  - Compute $s^{-1} \mod q$ *(the modular inverse of* s).
  - Recover the message:

$$m = c_2 \times s^{-1} \mod q$$

If an attacker can **efficiently solve** the discrete logarithm problem, they can **completely break** El Gamal encryption even in a passive **Eavesdropping Attack (EAV)** — where the attacker just observes the public key and intercepted ciphertext.

Here are the **steps**:

Step 1: Recover the Private Key

Given the public key (q, g, h):

- Solve for the private key x by computing:

$$x = \log_g(h) \mod q$$

(using the efficient discrete log algorithm available in PPT). Thus, the attacker now knows the secret key x, which normally only the receiver should know.

Step 2: Intercept the Ciphertext

- The attacker passively observes the ciphertext (c_1, c_2) being transmitted.
- Recall:
  - $c_1 = g^y \mod q$
  - $c_2 = m \times h^y \mod q$

Step 3: Decrypt the Ciphertext

Now that the attacker knows x, they can:

1. Compute:

$$s = c_1^x \mod q$$

(same as the shared secret in normal decryption).

2. Compute the modular inverse of s, denoted s^{-1}.
3. Recover the message m using:

$$m = c_2 \times s^{-1} \mod q$$

Thus, the attacker **decrypts the message completely**, without ever needing the legitimate private key holder's cooperation.

Question 4:

Why is the RSA-OAEP Public Key Encryption Scheme a Randomized Scheme?

Background:

RSA-OAEP stands for RSA Optimal Asymmetric Encryption Padding. It is an enhancement over plain RSA encryption to achieve semantic security (i.e., even if an attacker sees multiple ciphertexts, they learn nothing about the underlying plaintexts). It introduces randomness into the encryption process, which is why it is called a randomized encryption scheme.

Understanding RSA-OAEP:

In **plain RSA encryption**, encryption is deterministic:

- Encrypt mmm by computing:

$$c = m^e \mod N$$

- Given the same m and public key, the ciphertext c is always the same.

This determinism is dangerous because:

- An attacker could simply encrypt all possible messages and compare ciphertexts (a **dictionary attack**).
- It leaks information about message patterns.

RSA-OAEP modifies this by adding **random padding** before encrypting:

- A random seed is generated during encryption.
- Two hash functions (called a mask generation function, MGF) are used to mix the seed and the message.
- The message is padded and randomized before RSA encryption.

Detailed steps:

- Generate a random string r (random seed).
- Use two hash functions (mask generation functions) to "mask" the message and the random string in a specific way.
- Apply the RSA encryption function to the resulting padded message.

The random string r ensures that the padded message changes **every time**, even if the original message m is the same.

- As a result:
    - Two encryptions of the same plaintext under the same public key will produce different ciphertexts.
    - Attackers cannot tell whether two ciphertexts correspond to the same plaintext.
- This randomness is critical for achieving semantic security against chosen plaintext attacks (CPA security).

Question 5:

Small Message Space Attack in Plain RSA and How RSA-OAEP Prevents It

Background:

In **Plain (Naïve) RSA** public key encryption:

- A message m is encrypted directly as:

$$c = m^e \mod N$$

- The public key is (N, e).
- The encryption is **deterministic**: Encrypting the same mmm always gives the same ciphertext c.

Small Message Space Attack in Plain RSA:

Suppose:

- The plaintext can only be one of two possible messages: m_1 or m_2.
- The attacker knows the public key (N, e).
- The attacker intercepts a ciphertext c.

Here are the steps:

Step 1: Attacker guesses the possible plaintexts

- Since there are only two possible messages, m_1 and m_2, the attacker can **precompute** the possible ciphertexts:
  - Compute:

$$c_1 = m^e{}_1 \mod N$$

  - Compute:

$$c_2 = m_2^e \mod N$$

Step 2: Attacker compares

- The attacker compares the intercepted ciphertext c to c_1 and c_2.
- If:
  - $c = c_1$, the plaintext is m_1.
  - $c = c_2$, the plaintext is m_2.

Thus, the attacker successfully determines which message was sent.

Why This Attack Works in Plain RSA:

- **Deterministic encryption**: Same message always gives same ciphertext.
- **Small message space**: Few possible messages, so guessing is practical.
- **No randomness**: Encryption output is fully determined by mmm and public key.

How RSA-OAEP Thwarts the Small Message Space Attack:

RSA-OAEP (Optimal Asymmetric Encryption Padding) modifies the encryption process by **randomizing** it:

In RSA-OAEP:

- Before applying RSA encryption, the plaintext mmm is first **padded with a random value**.
- The padding uses a **random seed** and **mask generation functions (MGFs)** to combine the plaintext and randomness.
- Then the padded message is encrypted using RSA.

Thus:

- Encrypting the same message twice **gives different ciphertexts** every time.
- Even if the attacker knows m_1 and m_2, they cannot just encrypt them and compare, because each encryption will produce a **different ciphertext** due to the random padding.

Therefore, in RSA-OAEP:

- Attacker cannot precompute ciphertexts for m_1 and m_2.
- Comparing ciphertexts is meaningless because each ciphertext is **randomly different**.
- Without the private key, the attacker cannot distinguish which message was encrypted.