Lab 10

A):

Area of Interest (AOI)
Each player has an "area of interest" around them—usually defined by a radius or a bounding box. The server sends the player updates only about objects (players, NPCs, items) within that area.

Example:
In an open-world RPG, a player only receives updates about enemies, players, or events within 100 meters.
The moment another player enters this area, the server starts sending their data to the observing client.

Benefits:
Reduces update frequency for far-away or irrelevant entities.
Highly scalable, only a small subset of the game world is actively synchronized for each player.
Can be dynamically adjusted for performance tuning.

Team or Group-Based Interest
Updates are restricted to group or team members rather than spatial proximity.

Example:
In a strategy game or battle royale with squads, you always receive updates about your teammates, even if they're far away.
Spectators or team coaches can have custom interest groups (e.g., view all players).

Benefits:
Supports non-spatial relevance, like communication between squad members or shared objectives.
Ensures tactical awareness for team gameplay.
Useful in scenarios with high player dispersion but strong logical grouping.

B):

A **Potentially Visible Set** is a precomputed list of visible areas or objects from a specific region or viewpoint within the game world.
Commonly used in 3D games to determine what a player could see from a given location.

How It Works:
The game world is divided into cells or regions.
During design or runtime, the game calculates which other regions are potentially visible from each cell.
When a player is in a cell, the server sends updates only for entities in that cell's PVS.

Benefits:
Highly efficient in complex environments like cities or dungeons.
Reduces unnecessary network traffic by excluding entities that are occluded.
Good for indoor or line-of-sight-limited games.

Static zones are fixed geographical areas in the game world. Players and entities are assigned to zones, and updates are sent to clients based on zone membership.
Each zone might represent a room, region, or quadrant of the map.
Entities in the same or adjacent zones share state updates.

How It Works:
The world is divided into zones at design time.
When a player enters a zone, they subscribe to updates in that zone.
The server sends only data for that zone to the player.

Benefits:
Simple to implement.
Works well in open-world games where zones align with natural boundaries.
Can be combined with Level of Detail for added efficiency.

C):
SQL Injection is one of the most common server-side attacks where an attacker exploits vulnerabilities in an application's input handling to inject malicious SQL queries into the server's database.

Example: A game might have a login system where the server accepts user input and constructs a query to verify these credentials against the database.
An attacker could enter a specially crafted input, such as: Username : Admin or 1=1.
Here, the part 1=1 is always true, so the query would return all users, potentially giving the attacker access to any account. The attacker might gain unauthorized access, or the attack could allow them to delete or modify data, steal sensitive information, or bypass authentication.

To prevent : Could implement input validation (username check only if expecting numbers and letters) or whitelist characters to avoid issues. Restrict the permissions of the database user the application uses, for example don't allow them to delete from your DB. Use parameterized queries to separate SQL from data. This way you could use placeholders and input is never inputted directly into an SQL query.

D):
Aim bot is an example of cheating in a multiplayer game. An aimbot is a software tool that aids players by automatically aiming their weapon at opponents with perfect accuracy, giving them an unfair advantage.

An aimbot is often a third-party program that hooks into the game client. It monitors the game's memory or uses screen-reading techniques to detect the positions of enemy players. The aimbot is able to automatically move the player's crosshair to an enemy player's head

or body when they come into view. This results in the player achieving impossible accuracy, with perfect headshots or kills, regardless of their real skill level.
To prevent aimbots you can implement server side hit detection and validation. Server-side hit detection involves verifying all actions (shots and kills) on the game server. The server checks if the shot originated from a valid position, whether the target was actually hit, and whether the timing aligns with the player's movement.

The server compares the data from the client (shot position, timing, and hitboxes) against predefined hitboxes and ensures that the actions align with the expected physics of the game. If the client reports an impossible event (perfect headshot from a distance that is unreasonable), the server can either reject the event or flag the player for review.

This helps prevent and manage aimbot softwares in games.