MCQ Revision

Secure by Design

- **Least Privilege**: assign only minimum required permissions
- **Minimise Attack Surface**: added functionality increases risk
- **Secure Defaults**: default configurations must be secure
- **Defence in Depth**: layered protections prevent single-point failure
- **Fail Securely**: errors must not reveal internal details
- **Avoid Security by Obscurity**: hidden ≠ secure
- **Fix Issues Correctly**: patch safely without side effects

Secure by Design

- Security must be integrated from the earliest stages (requirements)
- Security is not a feature — it must be built into the development lifecycle
- Use best practices, standards, and frameworks to guide design
- Identify attack surfaces, data flows, and insecure structures early

Secure by Design

- **Threat modelling**:
  - Diagram (DFDs), Identify threats, Mitigate, Validate
  - Threat Categorisation
  - Ranking and prioritisation
  - Mitigations

Passing Data to Subsystems

- Dynamic web apps pass data to subsystems (DBs, OS, libraries), often via constructed strings
- Meta-characters cause context switches: characters that act as control instructions, not data
- Injection flaws occur when the application passes unvalidated data to a subsystem
- Any user-controlled input can be dangerous: text fields, dropdowns, checkboxes, HTTP parameters
- SQL Injection lets attackers modify or add queries by manipulating input
- Half-implemented validation (“half-way security measures”) still leaves exploitable gaps

Passing Data to Subsystems

- **Classic SQLi techniques**: unescaped quotes, SQL comments (--), Boolean logic (OR 'a'='a'), char() encoding, hex strings
- **Error-based SQLi**: attackers use database error messages to learn table/column details
- Stored procedures are not automatically safe - vulnerable if concatenation still used
- Database permissions help but are only a secondary defence
- “Washing” input (escaping/stripping meta-characters) is error-prone and incomplete
- Prepared statements / parameterized queries are the most reliable SQLi prevention method

Cross-site Scripting

- XSS injects malicious code that the browser executes, targeting users, not the application
- Caused by missing/weak input validation and output encoding
- Impacts: session cookie theft, keystroke logging, redirection to malware, privilege escalation
- Reflected XSS: payload sent in request and “reflected” in response; often delivered via phishing links
- Stored (Persistent) XSS: payload saved on server (comments, posts, etc.); executes automatically for all viewers
- XSS is a meta-character problem - scripts run when characters are interpreted as HTML/JS instead of plain text

Cross-site Scripting

- Session hijacking via XSS occurs when injected scripts read document.cookie and send to attacker
- Attackers embed cookie-stealing scripts in stored posts or comments; victims load page → script executes
- HTML Encoding neutralizes dangerous characters (<, >, ", ', &), but must be context-awareSelective filtering (allow-list approach) is difficult; better handled by libraries like HTMLPurifier
- Prevention: HTML encode output, validate/strip input, use CSP, secure cookies (HttpOnly)
- XSS payloads can hide in attributes, event handlers, image tags, JavaScript URIs, or encoded/obfuscated forms

Access Control

- Access control (authorisation) mediates access to resources based on identity and defined policies
- Lack of proper access control can lead to unauthorised information disclosure, modification, or business-function abuse
- IDOR: attacker manipulates direct references to internal objects (IDs, filenames) to access/modify unauthorised data
- Vulnerabilities include lateral/vertical privilege escalation, forced browsing, and unprotected object references
- Attackers modify URLs, parameters, cookies, form fields, or POST data to access other users’ resources
- Developers often expose predictable identifiers (e.g., numeric IDs), enabling enumeration and unauthorised access

Access Control

- **Forced browsing**: accessing restricted pages by altering navigation parameters (e.g., fwd=admin.jsp)
- **Privilege escalation**: brute-forcing logins or abusing broken authentication to impersonate higher-privilege users
- **Path traversal**: using ../ to access files outside intended directories (e.g., /etc/passwd, hidden resources)
- **Detection**: test direct references by substituting values; responses reveal whether access controls are enforced
- **Defence**: allow-list input validation, indirect object references, strong authentication, rate-limiting, consistent error messages
- **Prevent path injection**: proper file permissions, avoid direct file references, centralise access control for consistent enforcement

CSRF

- CSRF forces a victim’s authenticated browser to perform unwanted actions on a trusted site
- Targets state-changing requests (e.g., delete user, transfer money), not data theft
- Works because browsers automatically include cookies (session IDs) in requests, even cross-site
- Attackers rely on victims being logged in and then trick them into making a crafted request
- Example vectors: hidden images, malicious links, auto-submitted forms on attacker-controlled pages
- Impact depends on victim’s privileges—admin victims allow full compromise of the application

CSRF

- GET-based CSRF: `<img src="https://site/delete?user=bob">` triggers privileged actions silently
- POST-based CSRF uses hidden forms auto-submitted to perform admin-level actions
- CSRF is possible when apps rely only on cookies for authentication and have no request-origin validation
- Primary defence: Synchronizer Token Pattern: unique per-session/per-request CSRF tokens validated server-side
- Additional protections: SameSite cookie flags (Strict, Lax, None), built-in CSRF middleware (e.g., Django)
- CSRF still possible if apps allow state-changing GET requests or deliberately set cookies to `SameSite=None`

Input Validation

- Input validation ensures applications accept only correctly formatted, strongly typed, and bounded data
- All external input is untrusted: URLs, forms, cookies, headers, files, and server-generated input
- Validation must be applied at every tier, as early as possible, using both syntactic (format) and semantic (business logic) checks
- Language behaviours (ASP Request shortcuts, PHP variable mapping) can let attackers override internal variables
- Input validation should verify lengths, ranges, types, syntax, and detect tampering vs. user mistakes
- Prefer allowlisting (accept known good) over blocklisting due to infinite attack variants

Input Validation

- Sanitisation transforms data into a safe format; allowlist sanitisation is safer (remove/encode all non-approved characters)
- Regular expressions are powerful for pattern validation; OWASP provides RE repositories for common fields
- Client-side validation improves UX but cannot be trusted for security—server-side checks are mandatory
- Applications must log suspicious input; server logs alone miss POST parameters and attack context
- Avoid exposing internal state (IDs, flags) to clients; use integrity checks like hashes for server-generated data
- Don’t rely on security by obscurity—hidden URLs, folders, or client-side checks do not protect data or functions

Secure Code Analysis

- Secure code review identifies insecure code & root causes before deployment; complements peer reviews
- Manual + automated analysis is best: tools find common flaws, humans verify context, logic, business rules
- AI-assisted code review (Copilot, CodeWhisperer, JetBrains AI) can detect logic flaws & suggest fixes but still needs human oversight
- SAST = static analysis of non-running source code (white-box); DAST = testing a running app (black-box)
- SAST finds hard-to-reach flaws early ("shift left") and supports secure SDLC by analysing code structure & data flows
- SAST techniques include control flow graphs, basic blocks, taint analysis, and binary analysis for code without source access

Secure Code Analysis

- SAST pros: scalable, fast, good at “low-hanging fruit” (SQLi, buffer overflows); cons: false positives, misses logic/integrity issues
- False positives = reported issues that aren’t exploitable; false negatives = real vulnerabilities tools fail to detect
- Modern secure tooling includes Snyk Code, SonarQube, Checkmarx, Semgrep, GitHub Advanced Security, CodeQL
- IaC static analysis (Terraform, Kubernetes, Dockerfiles) is essential due to cloud misconfiguration risks
- SCA (Software Composition Analysis) tracks open-source libraries, SBOM, licenses, and known vulnerabilities
- Supply chain security now critical: dependency confusion, malicious packages, poisoned updates → require provenance checks (Sigstore, Cosign)

Summary of study material

- **Security by Design**: least privilege, secure defaults, defence-in-depth, fail securely, minimising attack surface
- **Threat Modelling**: STRIDE, DFDs, trust boundaries, attack surface identification
- **Input & Data Validation**: allow-listing vs block-listing, sanitisation, regex validation
- **Injection Attacks**: SQLi, Command Injection, meta-characters, prepared statements/parameterised queries

Summary of study material

- **XSS**: reflected vs stored, where user input is echoed, cookie/session theft, HTML escaping, HttpOnly
- **CSRF**: SameSite cookie behaviour, CSRF tokens, why GET must not modify state
- **Access Control**: IDOR, forced browsing, privilege escalation, path traversal, indirect object references
- **Secure Code Testing**: Secure Code Review, SAST vs DAST, SAST Techniques: dataflow/taint analysis, control-flow graphs, basic blocks, binary analysis, Software Composition Analysis, Security Features vs Security.