CISSP

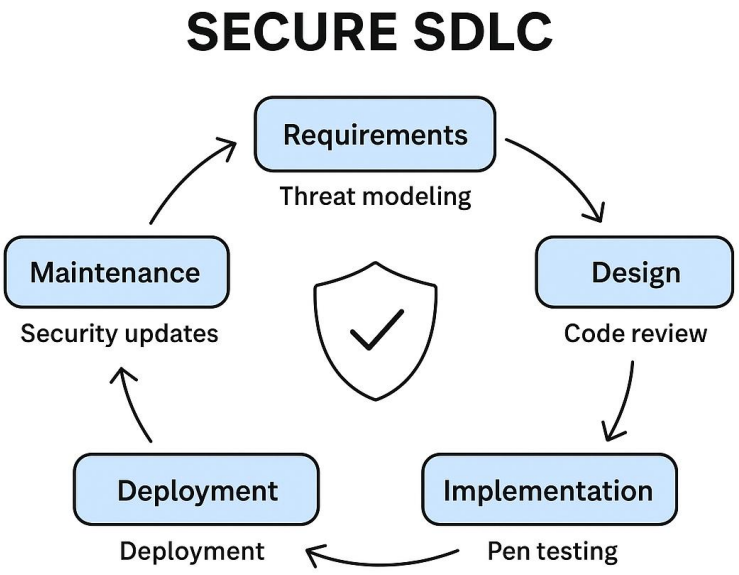
LAST MINUTE STUDY GUIDE

DOMAIN 8
SOFTWARE DEVELOPMENT SECURITY

Section 1 — Understand & Integrate Security into the SDLC

1.1 Why security in the SDLC matters

- **Goal:** Move security from an afterthought to an integral part of each phase of development so defects are prevented early (cheaper to fix) and systems are resilient by design.
- **Business drivers:** reduce breaches, compliance (GDPR/PCI/HIPAA), protect IP, reduce incident cost, faster time-to-remediation.
- **Cost of late fixes:** Defects found in production can cost 10–100× what they cost during design or coding.
- **Mindset:** Shift-left security — find & fix earlier; DevSecOps — "security as code".



1.2 SDLC models — summary + security implications

Describe model, pros/cons, where to add security.

1. **Waterfall (Linear)**
    - Phases: Requirements → Design → Implementation → Verification → Maintenance.

- Security implications: Insert formal security gates at each phase (security requirements, design review, SAST during implementation, DAST before release).
- When used: Regulated environments or legacy projects.

2. **V-Model (Verification & Validation)**
    - Each development phase maps to a test phase.
    - Security: Map security tests to each corresponding phase (e.g., threat model ↔ architecture review).
3. **Spiral**
    - Iterative with risk analysis each cycle.
    - Security: Use cycle-based threat modeling and risk reprioritization.
4. **Agile (Scrum/Kanban)**
    - Short iterations, frequent releases.
    - Security: Embed security user stories, "Definition of Done" includes security checks, lightweight threat modeling per story/epic.
5. **DevOps / DevSecOps**
    - Continuous Integration / Continuous Delivery (CI/CD).
    - Security: Automate SAST/DAST/Dependency scans, code-signing, secrets management, configuration hardening via IaC.
6. **RAD / XP / Prototyping**
    - Rapid iterations, early prototypes.
    - Security: Focus on secure prototyping practices, ephemeral test environments, strong access control to avoid leakage.

1.3 Security activities mapped to SDLC phases

A. Requirements Phase

- **Deliverables:** Security & privacy requirements, compliance mapping, acceptance criteria.
- **Activities:**

- Identify functional and non-functional security requirements (availability, integrity, confidentiality).
- Legal & regulatory mapping: GDPR data processing, PCI data flows, HIPAA PHI handling, industry-specific controls.
- Data classification: classify data types used/produced by application (PII, sensitive, public).
- Threat surface definition: assets, trust boundaries, actors, entry points.
- Define security acceptance tests (what must pass before release).

- **Artifacts:** Security requirements document, data flow diagrams (DFDs), privacy impact assessment (PIA), compliance checklist.
- **Checklist:**
  - Have you documented compliance obligations?
  - Are security requirements measurable/testable?
  - Are privacy-by-design requirements defined?

B. Architecture & Design Phase

- **Deliverables:** Secure architecture diagrams, threat model, security control matrix.
- **Activities:**
  - Threat modeling (STRIDE/PASTA/Attack trees) — identify misuse cases and mitigation mapping.
  - Define authentication and authorization architecture (OAuth2/OIDC, RBAC/ABAC).
  - Session design: session lifetime, token revocation, secure cookie flags.
  - Data protection: encryption at rest (TDE, disk/DB-level), encryption in transit (TLS), key lifecycle.
  - API design: pagination, rate limit, input validation, secure default headers.
  - Secure dependency strategy: accepted third-party libs, SBOM requirements.
  - Design for failure: timeouts, circuit breakers, retry logic.
  - Logging & telemetry design: what to log (no sensitive data), format, correlation IDs.

- Secrets & config management design: where to store secrets (Vault, KMS), rotation policy.

- **Artifacts:** Architecture decision records, threat model outputs, component security checklist, sequence diagrams.
- **Example tasks:**
  - Diagram trust boundaries and data stores; label sensitive flows.
  - Map STRIDE threats to architecture components and assign mitigations.

C. Implementation / Coding Phase

- **Deliverables:** Source code following secure coding standards, unit tests, SAST reports.
- **Activities:**
  - Adopt secure coding standards (OWASP, SEI CERT) with concrete rules.
  - Code reviews and pair programming with security checklist.
  - Apply static code analysis (SAST) integrated into CI pipeline — fail builds on high/critical findings.
  - Secrets detection in code (prevent accidental commits).
  - Dependency scanning (SBOM, CVE matching).
  - Defensive coding: input validation, output encoding, parameterized DB access, safe deserialization patterns.
  - Memory safety practices in native languages (bounds checking, avoid unsafe APIs).
- **Tools:** SonarQube, Checkmarx, ESLint/TSLint with security rules, Bandit for Python, spotbugs/findsecbugs for Java.
- **Checklist for devs:**
  - No hard-coded credentials or keys.
  - Inputs validated / sanitized.
  - Use prepared statements for DB queries.
  - Appropriate error handling (no stack traces in production).

D. Testing / Verification Phase

- **Deliverables:** DAST reports, penetration test reports, SCA (software composition analysis) results, security regression tests.
- **Activities:**
    - Dynamic testing (DAST) against running app: injection/XSS detection, logic vulns.
    - Penetration testing (black/gray/white box) on major releases.
    - Fuzzing for input handling modules (AFL, Peach).
    - IAST for runtime feedback (helps find vulnerability in context).
    - Security-focused unit/integration tests; include negative tests.
    - Regression testing to ensure fixes remain effective.
- **Tools:** Burp Suite, OWASP ZAP, Nikto, Arachni, Cuckoo for payload testing.
- **Checklist:**
    - Have critical flows been fuzzed?
    - Are authentication/authorization rules tested end-to-end?
    - Are all CVEs in dependencies addressed or mitigated?

E. Release / Deployment Phase

- **Deliverables:** Signed artifacts, hardened configs, deployment runbooks.
- **Activities:**
    - Secure build pipeline: sign artifacts, verify checksum, enforce pipeline policies.
    - Hardening of runtime environment: minimal OS footprint, up-to-date libraries, disable unused services.
    - Container security: scanned images (Clair/Trivy), minimal base images, immutable tags.
    - Infrastructure as Code (IaC) validation (Checkov/terrascan).
    - Secrets injection at runtime via secure stores.
    - Canary/blue-green deployment for safe rollout.
- **Checklist:**
    - Are images signed and scanned?

- Is roll-back plan defined?
- Are runtime monitoring agents enabled?

F. Maintenance & Operations Phase

- **Deliverables:** Monitoring rules, patch management records, incident playbooks.
- **Activities:**
  - Continuous monitoring: logs to SIEM, alerting on suspicious patterns.
  - Vulnerability management: periodic SCA and patching.
  - Patch testing in staging and canary before prod rollout.
  - Incident response integration and playbooks for app-level incidents.
  - Periodic re-threat-modeling for major changes.
  - Deprecation and secure decommissioning plans.
- **Checklist:**
  - Are critical CVEs patched within SLA?
  - Are logs retained per policy and protected?
  - Are backups tested and encrypted?

1.4 Threat Modeling — exhaustive how-to & patterns

- **Purpose:** Systematically identify threats, likelihood, and controls.
- **Models/approaches:**
  - **STRIDE:** Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege.
  - **PASTA:** Process-oriented — aligns business risk with technical scenarios.
  - **VAST / Trike:** Scalable for many teams.
  - **Attack Trees:** Graphical representation of attacker goals and methods.
- **Process (practical steps):**

1. **Define scope:** what system, data, and interfaces.
2. **Create data flow diagrams (DFDs):** show processes, data stores, external entities, trust boundaries.
3. **Identify assets & trust boundaries.**

4. **Enumerate threats** using STRIDE per DFD element.
5. **Assess likelihood and impact** — qualitative or quantitative (CVSS-like).
6. **Prioritize threats** and design mitigations.
7. **Document residual risk** and acceptance criteria.

- **Artifacts:** DFDs, threat listing table (threat, actor, asset, mitigations, owner).
- **Tooling:** Microsoft Threat Modeling Tool, OWASP Threat Dragon.
- **Tip:** Do threat modeling early and repeat after design or major changes.

1.5 Defining measurable security requirements

- Translate high-level security goals into measurable requirements:
    - Example: “All sensitive data in transit must use TLS 1.2+ with strong ciphers” (measurable).
    - “Password storage must use Argon2id with parameters X,Y,Z” (testable).
    - “All third-party libraries must have no critical CVEs or compensating controls in place” (policy & automation).
- Use **RACI** to assign responsibility.

1.6 Secure design patterns & anti-patterns

- **Patterns to use:**
    - Secure gateway (API gateway with auth & throttling).
    - Circuit breaker for resilience.
    - Token-based session management with revocation lists.
    - Encrypted data vaults for PII/keys.
    - Defense-in-depth: input validation, service-layer authorization, DB access control.
- **Anti-patterns to avoid:**
    - Security through obscurity.
    - Single-tier monolithic with DB credentials in code.
    - Relying solely on perimeter controls.

1.7 Privacy & data protection in SDLC

- **Privacy by Design**: minimize data collection, anonymize, pseudonymize, implement retention & deletion policies.
- **Data flows mapping** for GDPR DPIA (Data Protection Impact Assessment).
- **Consent & lawful basis** baked into requirements for personal data.
- **Cross-border data flows** and contractual clauses for processors/sub-processors.

1.8 Secure DevOps / DevSecOps practices

Principles: automate security gates, integrate tests into pipelines, shift-left, security as code.

- **CI/CD security controls:**
  - Pipeline credential isolation (short-lived tokens).
  - Pipeline policy as code (e.g., GitOps with signed manifests).
  - Mandatory SAST on PRs, DAST in pre-prod, SCA on each build.
  - Policy enforcement: prevent merge if high-risk findings.
- **Secrets management:** HashiCorp Vault, AWS Secrets Manager, Azure Key Vault; never in repo.
- **Artifact management:** use registries (Artifactory, Nexus) with access control & immutability.

1.9 Software supply chain security

SBOM (Software Bill of Materials): list of third-party components and versions.

- **Risks:** malicious packages, typosquatting, compromised CI tools, compromised mirrors.
- **Controls:**
  - Enforce SBOM generation for builds.

- SCA (Software Composition Analysis) tools: Snyk, WhiteSource, OWASP Dependency-Check.
- Verify signatures of third-party binaries, code signing enforcement.
- Harden build environments (least privilege, immutable build servers).
- Vet suppliers: security questionnaires, pen-tests, SLA clauses, right-to-audit.

- **Incident process:** vuln disclosure path, emergency patching, dependency replacement plan.

1.10 Testing strategies, tools & automation

- **Testing pyramid:** Unit tests (fast), integration tests, system tests, UI tests (slow). Add security tests across pyramid.
- **SAST (Static):** checks code before runtime.
  - Strengths: finds hard-coded credentials, injection patterns.
  - Limitations: false positives, no runtime context.
- **DAST (Dynamic):** black-box testing on running app.
  - Strengths: finds runtime issues like auth bypasses, session misconfigurations.
  - Limitations: limited code insight, may miss logic bugs.
- **IAST (Interactive):** agent on the app during tests — combines SAST+DAST strengths.
- **RASP (Runtime Application Self-Protection):** app-level monitoring and blocking in runtime.
- **Fuzzing:** automated random input to find parser bugs.
- **Penetration Testing:** manual, explores business-logic issues.
- **Security regression tests:** every fix must have test added to prevent recurrence.
- **Tool chaining in CI/CD:** SAST -> Build -> SCA -> DAST -> Pre-prod acceptance.

1.11 Metrics & KPIs for SDLC security

- **Preventive metrics:** % of projects with threat model, % of features with security requirements.
- **Detective metrics:** # of vulnerabilities per KLOC, SAST/DAST false positive rate.
- **Remediation metrics:** Mean time to remediate vulnerabilities (MTTR), % raised vs fixed per release.
- **Process metrics:** % of builds failing due to policy violations, % of commits with secrets detected.
- **Quality metrics:** % code coverage for security tests, % of PRs with SAST run.

1.12 Governance, roles & responsibilities

- **Roles:**
    - Product Owner: ensures security requirements in backlog.
    - Dev Lead: enforces coding standards & code reviews.
    - Security Champion: embedded in dev teams for first-line security support.
    - AppSec team: provides threat models, tools, approval gates.
    - CI/CD owner: ensures pipeline security and artifact integrity.
- **RACI examples:** who is Responsible, Accountable, Consulted, Informed for each security gate.
- **Security policies:** coding standards, dependency policy, release gating.

1.13 Documentation & evidence for audits

- Maintain artifacts: threat models, SAST/DAST reports, SBOM, signed build logs, test results, change logs, deployment approvals.
- For audits: exportable evidence, immutable storage, retention per policy.

1.14 Common pitfalls & how to avoid them

- **Pitfall:** “We’ll fix security later.” → Avoid with policy: no release without passing security gate.
- **Pitfall:** Over-reliance on a single tool. → Use SAST+DAST+SCA combined.

- **Pitfall:** Secrets in repos. → Enforce pre-commit hooks & detection.
- **Pitfall:** Unmonitored third-party libs. → Maintain SBOM, automated alerts for new CVEs.
- **Pitfall:** No rollback or chaos planning. → Build safe rollback & blue/green deployments.

1.15 Practical checklists / templates

Threat Modeling Quick Table (template)

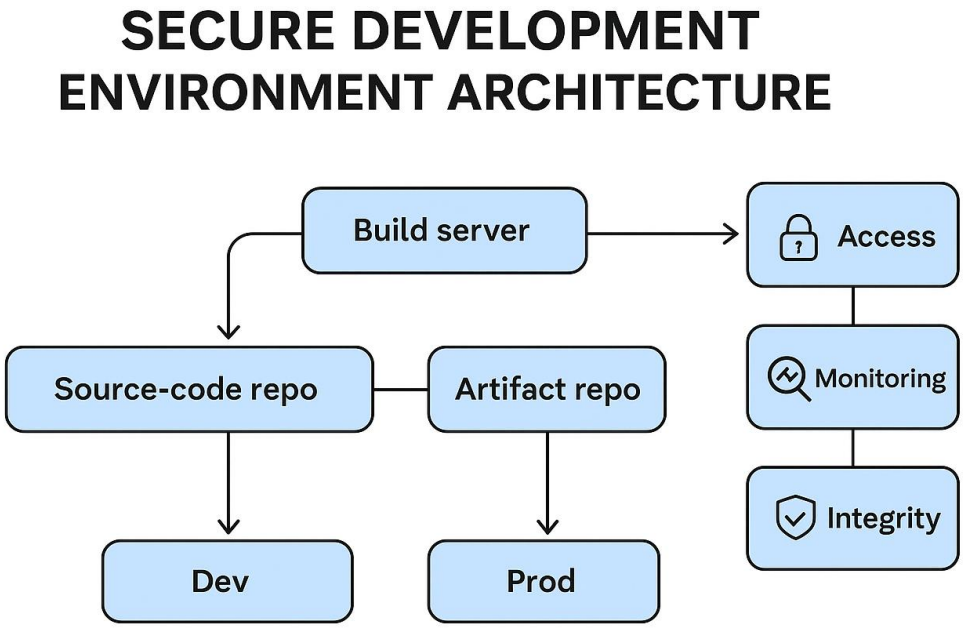
Item	Details
System	e.g., Payments Service v2
Data Assets	e.g., Card PAN (sensitive), tokens
Trust Boundaries	e.g., Client ↔ API Gateway, Internal Network ↔ DB
Actors	e.g., Authenticated user, Admin, External Partner
STRIDE Threat	e.g., Tampering — Data tampering via API
Mitigation	e.g., HMAC signatures, input validation, rate limiting
Residual Risk	e.g., Acceptable with logging & alerting
Owner	e.g., Payment Service Owner

Secure Deployment Gate (release checklist)

- SAST high/critical findings resolved or accepted by risk owner.
- DAST scans completed; no OWASP Top 10 critical in prod flows.
- SBOM produced and low/no critical CVEs.
- Artifacts signed; checksums verified.
- Secrets not present in artifacts.
- Rollback plan and monitoring in place.
- Compliance evidence packaged.

SECTION 2 — Secure Development Environment (SDE)

A Secure Development Environment ensures that code, tools, and developers operate within a controlled, monitored, and protected workspace, minimizing unauthorized access, code tampering, or data leaks.



2.1 Key Principles

- **Confidentiality:** Protect source code, credentials, and design documents.
- **Integrity:** Prevent unauthorized modifications to code or builds.
- **Availability:** Ensure critical development resources (build servers, repos) are resilient.
- **Accountability:** Every developer action should be attributable and auditable.

2.2 Environment Components

Environment	Purpose	Security Focus
Development	Code creation & unit testing	Least privilege, secure coding, source control
Test / QA	Validation before release	Data masking, controlled access

Staging / Pre-prod	Final acceptance	Same config as prod, but isolated
Production	End-user deployment	Change control, monitoring, patching

- Strict segregation between dev → test → prod is crucial.
- Use different credentials for each; no shared accounts.

2.3 Secure Coding Workstations

- Hardened OS with endpoint protection, disk encryption, and minimal privileges.
- Disable USB/media access (to prevent IP theft).
- Require VPN with MFA for remote devs.
- Ensure patching and automatic updates.
- Enforce security baseline via MDM or configuration management (Intune, Jamf, Ansible).

2.4 Source Code Repository Security

- Examples: GitHub Enterprise, GitLab, Bitbucket.
- Use MFA for all access.
- Restrict merge permissions; enforce code review policy.
- Apply branch protection rules and signed commits (GPG).
- Regularly scan repos for secrets (TruffleHog, GitGuardian).
- Audit commit history and repository settings periodically.

2.5 Build and Integration Server Security

- Tools: Jenkins, GitHub Actions, Azure DevOps, GitLab CI/CD.
- Run build agents in isolated, ephemeral containers.
- Sign build artifacts (using GPG or Sigstore).
- Store secrets in encrypted vaults (never plaintext in pipelines).
- Limit build server privileges — avoid direct production access.

- Verify build integrity using checksums and hash validation.

2.6 Configuration and Dependency Management

- Lock dependency versions (use “package-lock.json,” “requirements.txt”).
- Regularly perform Software Composition Analysis (SCA).
- Maintain SBOM (Software Bill of Materials) for transparency.
- Automatically block outdated or vulnerable dependencies.

2.7 Monitoring and Logging in SDE

- Centralized logging of repository access, build triggers, and deployment actions.
- Detect anomalies (e.g., commits from unusual geolocations).
- Integrate logs into SIEM (e.g., Splunk, ELK).

2.8 Legal and Compliance Considerations

- Respect licensing (open-source GPL, Apache, MIT).
- Enforce copyright and IP controls.
- Keep records of third-party software versions for audit trails.

SECTION 3 — Software Security Testing and Assessment

3.1 Purpose

To validate that the software meets security objectives and does not contain exploitable flaws — throughout development and maintenance.

3.2 Testing Categories

Type	Description	Tools / Focus
Static Testing (SAST)	Examines code before execution	Checkmarx, SonarQube, Fortify

Dynamic Testing (DAST)	Tests running app externally	OWASP ZAP, Burp Suite
Interactive Testing (IAST)	Combines SAST + DAST via agent	Contrast Security, Seeker
Runtime Application Self-Protection (RASP)	Monitors and blocks attacks in runtime	Signal Sciences, Imperva
Fuzz Testing	Random input to find memory or logic flaws	AFL, Peach, BooFuzz
Penetration Testing	Manual, logic-focused attack simulation	Offensive teams, third-party testers
Regression Testing	Ensures previously fixed flaws remain resolved	Automated suites
Software Composition Analysis (SCA)	Identifies vulnerable dependencies	Snyk, WhiteSource, Dependency-Check

3.3 Testing Phases Integration

- **Unit Testing:** Verify security at function level (input validation, error handling).
- **Integration Testing:** Verify data flow between modules, interface security.
- **System Testing:** Test full environment — authentication, encryption, logging.
- **Acceptance Testing:** Validate security requirements before release.
- **Post-Deployment Testing:** Monitor for zero-day vulnerabilities and misconfigurations.

3.4 Common Vulnerabilities to Test

- **OWASP Top 10:** Injection, Broken Auth, Sensitive Data Exposure, XSS, SSRF, Security Misconfiguration.
- **CWE/SANS Top 25:** Unsafe deserialization, improper error handling, buffer overflow.

3.5 Secure Test Data Management

- Use synthetic data or masked production data in test environments.
- Encrypt all test data at rest.

- Ensure data disposal post-testing (no PII leakage).

3.6 Reporting and Metrics

- Report severity, impact, likelihood.
- Classify issues: Critical → must fix before release; Medium → track and plan; Low → accept or monitor.
- Maintain trend metrics:
    - Mean time to detect (MTTD)
    - Mean time to remediate (MTTR)
    - Vulnerability recurrence rate

3.7 Continuous Testing in CI/CD

- Embed SAST/DAST/SCA in the pipeline.
- Automate “build breaks” on critical findings.
- Generate security scorecards for every build.
- Require risk sign-off before promotion to production.

SECTION 4 — Software Deployment, Release, and Maintenance

4.1 Secure Release Management

- **Release Authorization:** Only approved builds move to production (change management approval).
- **Artifact Signing:** Validate authenticity using cryptographic signatures.
- **Version Control:** Maintain release tagging and rollback capability.
- **Rollback Plan:** Pre-approved reversion plan for deployment failures.

4.2 Deployment Security

- **Environment Hardening:** Disable default accounts, change default passwords.

- **Secure Configuration Baseline:** Harden OS, app, and middleware (e.g., CIS Benchmarks).
- **Infrastructure as Code (IaC) Validation:** Tools like Checkov, TerraScan for security compliance.
- **Secrets Injection:** Only at runtime from secure vaults.
- **Secure APIs & Endpoints:** Use HTTPS, certificate pinning, proper CORS configuration.
- **Cloud Deployment Controls:** IAM roles, least privilege, network segmentation, and encryption enforced by templates.

4.3 Patch and Vulnerability Management

- **Patch Lifecycle:**
    1. Identify new vulnerabilities
    2. Evaluate severity and exploitability
    3. Test patch in staging
    4. Deploy during maintenance windows
    5. Verify and document
- **Vulnerability Scanning Tools:** Nessus, Qualys, OpenVAS.
- **Prioritization:** Based on CVSS score and asset criticality.

4.4 Change and Configuration Control

- Formal Change Control Board (CCB) approves updates.
- Track all changes via tickets (JIRA/ServiceNow).
- Maintain configuration baseline documentation.
- Use version control for infrastructure (GitOps).
- Detect configuration drift using automation (Ansible, Chef).

4.5 Monitoring and Maintenance

- Implement continuous monitoring of deployed software: performance, security, and user behavior.
- Monitor for indicators of compromise (IoCs).
- Integrate alerts into SOC/SIEM.
- Conduct periodic security audits and log reviews.

4.6 Secure End-of-Life (EOL) and Decommissioning

- **Retirement Policy:** Define criteria for product end-of-life.
- **Data Sanitization:** Wipe storage media (NIST SP 800-88).
- **Revoke access keys, certificates, tokens** associated with legacy systems.
- **Customer Notification:** For SaaS or externally deployed products.
- **Archival:** Retain minimal records for legal or audit obligations only.

SECTION 5 — Secure Coding Principles and Practices

5.1 Secure Coding Overview

Secure coding ensures that software behaves predictably under both normal and malicious conditions. Poor coding decisions can introduce exploitable vulnerabilities.

Key frameworks and references:

- **OWASP Top 10** – Common web-app flaws.
- **CWE (Common Weakness Enumeration)** – Catalog of coding weaknesses.
- **CERT Secure Coding Standards** – Language-specific guidelines (C, C++, Java).
- **ISO/IEC 27034** – Application security framework.

5.2 Core Principles

Principle	Description	Example
Input Validation	Validate *everything* coming from users, APIs, or files.	Use whitelisting for expected formats.

Output Encoding	Encode data before displaying to prevent XSS.	htmlspecialchars() in PHP.
Error Handling	Avoid detailed error messages; log securely.	Show “Invalid input” instead of stack trace.
Least Privilege	Run processes with minimal permissions.	App runs under limited user, not admin.
Fail Securely	Default to secure state on failure.	If auth server fails, deny access.
Defense in Depth	Multiple overlapping controls.	WAF + input validation + output encoding.
Secure Defaults	Preconfigure secure settings.	“Account lockout = enabled.”
Don’t Trust Client	Enforce validation on server side.	Server re-verifies form input.
Session Management	Use secure cookies, regenerate IDs, set expiry.	HTTPS + HttpOnly + SameSite cookies.
Cryptography	Use tested libraries and current algorithms.	AES-256, RSA-2048, SHA-256.

5.3 Language-Specific Concerns

- **C/C++:** Buffer overflows, pointer misuse → use bounds checking.
- **Java:** Deserialization attacks, reflection misuse → disable unsafe features.
- **Python:** Insecure eval, pickle loads → use ast.literal_eval.
- **JavaScript:** XSS, prototype pollution → sanitize inputs, freeze objects.

5.4 Secure Authentication & Authorization

- Use multi-factor authentication.
- Hash passwords with strong algorithms (bcrypt, Argon2).
- Use role-based access control (RBAC) or attribute-based (ABAC).
- Verify access at every request (not just login).

5.5 Secure Data Handling

- Encrypt data at rest and in transit (TLS 1.2+).
- Apply data classification and retention policies.
- Use secure key management (HSM, Vault).
- Zeroize sensitive data in memory after use.

5.6 Input Validation Patterns

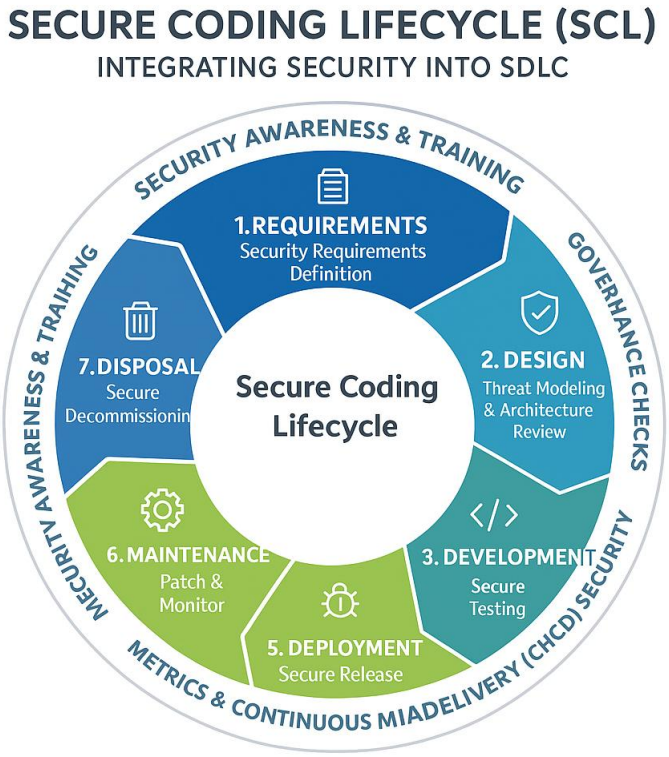
- Positive validation preferred (define allowed patterns).
- Reject known bad patterns (blacklisting) is weak.
- Normalize input before validation.

5.7 Common Coding Vulnerabilities

Vulnerability	Description	Example
SQL Injection	Unsanitized SQL input	' OR 1=1--
XSS	Unencoded user input rendered in browser	<script>alert()</script>
CSRF	Forged cross-site requests	Hidden form auto-submits request
Insecure Deserialization	Untrusted object input	Deserialize manipulated object
Path Traversal	Manipulate file path	../../etc/passwd
Command Injection	Inject shell commands	os.system("rm " + filename)

5.8 Secure Coding Lifecycle

1. Establish coding standards.
2. Train developers.
3. Perform peer code reviews.
4. Automate static analysis (SAST).
5. Remediate findings promptly.



SECTION 6 — Software Acquisition, Outsourcing, and Third-Party Development

6.1 Secure Acquisition Process

Software can be built, bought, or outsourced — all require due diligence.

Steps:

1. **Requirements Definition:** Include security and compliance needs (e.g., GDPR, PCI).
2. **Vendor Evaluation:** Review vendor's security posture, certifications (ISO 27001, SOC 2).

3. **Contract Clauses:**
    - Security requirements
    - Right to audit
    - Breach notification timelines
    - Source-code escrow
4. **Security Review:** Before acceptance, perform vulnerability scans/pen tests.

6.2 Open-Source Software (OSS) Considerations

- Review license terms (GPL vs MIT).
- Use trusted repositories only.
- Continuously scan for vulnerabilities (SCA).
- Track components in SBOM for transparency.

6.3 Third-Party & Outsourced Development Risks

Risk	Control
Loss of IP	NDA, restricted access
Malicious code	Code reviews, scanning
Inconsistent security	Shared policies, audits
Lack of visibility	Logging, monitoring, regular reporting
Data leakage	Secure VPN, DLP, isolated environment

6.4 Secure Code Delivery

- Transfer source/binaries through encrypted channels.
- Verify digital signatures.
- Use hash verification (SHA-256 checksum).

SECTION 7 — Supply Chain and Software Integrity Security

7.1 Concept

Supply-chain security ensures that every component — from libraries to deployment infrastructure — maintains integrity and is free from compromise.

7.2 Threats

- **Dependency poisoning:** Malicious code in upstream packages.
- **Typosquatting:** Fake packages (e.g., "reqeusts" instead of "requests").
- **CI/CD pipeline compromise:** Attackers tamper with build servers.
- **Hardware/firmware backdoors:** In malicious drivers or devices.

7.3 Controls

Category	Control
Source Integrity	Sign commits, verify maintainer keys
Build Security	Isolate build environments, verify hashes
Artifact Signing	Use Sigstore, GPG for code signing
Dependency Management	SBOM, lockfiles, allow-listed registries
Runtime Protection	Verify signed binaries before execution
Monitoring	Detect anomalies in supply-chain pipeline
Vendor Vetting	Periodic security assessments

7.4 Supply-Chain Frameworks

- **NIST SP 800-161 Rev 1** – Supply-chain risk management.
- **SLSA (Supply-chain Levels for Software Artifacts)** – Integrity maturity model.
- **ISO 28000** – Supply-chain security management system.

7.5 Incident Example

SolarWinds Orion attack:
Hackers injected malware into signed updates distributed to customers — classic supply-chain compromise.

SECTION 8 — Governance, Security, and Compliance in SDLC

8.1 Objective

Ensure that security is managed as a process, not as a one-time control, by integrating policies, compliance, and assurance across the Software Development Life Cycle (SDLC).

8.2 Secure SDLC Models

Model	Description	Security Integration
Waterfall	Sequential stages	Add security gates between phases
Agile	Iterative, fast releases	Embed security in every sprint (DevSecOps)
DevOps / DevSecOps	Continuous delivery	Automated testing, monitoring, compliance
Spiral / V-Model	Iterative refinement	Security checks each iteration

8.3 Security Roles

- **Developers:** Implement secure code.
- **Security Champions:** Embed security in agile teams.
- **AppSec Engineers:** Conduct reviews and testing.
- **Auditors/Compliance Officers:** Ensure adherence to standards.

8.4 Security Metrics & KPIs

- % of builds passing security checks.
- Mean time to fix vulnerabilities.
- of security regressions per release**.**
- Compliance audit pass rate.

8.5 Policies and Standards

- Secure Coding Policy
- Change Management Policy
- Access Control Policy
- Vulnerability Management Policy

These ensure consistency and accountability in the SDLC.

8.6 Compliance Requirements

- **PCI-DSS:** Secure coding, patching, encryption.
- **GDPR / Data Protection Acts:** Privacy by design.
- **SOX / HIPAA:** Integrity, auditability, confidentiality.
- **ISO 27001 / 27034:** Information & application security management.

8.7 Continuous Improvement

- Conduct regular post-release reviews.
- Feed security learnings back into requirements and design.
- Foster security culture — “shift left” mindset.

THANK YOU

Enroll with MoS – **CISSP Training @ ₹5,999!**

WWW.MINISTRYOFSECURITY.CO