CSSLP

Certified Secure Software Lifecycle Professional

ISC2 Certification

CSSLP **Notes**

Murugesan Kandasamy

Domain 1 - Secure Software Concepts (12%)

1.1 Understand Core Concepts

These are the foundational principles, often summarized by the **CIA** triad (Confidentiality, Integrity, Availability) and the **AAA** triad (Authentication, Authorization, Accountability).

1.1.1 Confidentiality (C)

Definition: Protecting data and information from unauthorized disclosure or unauthorized access. The goal is **privacy** and **secrecy**.

1.1.1.1 Encryption

- **Core Concept:** A mathematical process that transforms plaintext data into an unreadable ciphertext, ensuring that only authorized parties with the correct key can access the original data.
- **Software Application Context:** Used to protect data **in transit** (e.g., using TLS/SSL) and **at rest** (e.g., encrypting database fields).
- **Use Cases/Considerations:**
    - **Use When:** Handling sensitive data (e.g., PII, financial, intellectual property).
    - **Not Use When:** Only speed is a concern for low-sensitivity data, as encryption/decryption adds overhead.
    - **Pros:** Protects data even if the storage system or network is compromised.
    - **Cons:** Requires robust and complex **key management** to protect the decryption keys.

1.1.1.2 Access Control

- **Core Concept:** Mechanisms (policies and rules) that limit which users, processes, or systems can view sensitive data. This is enforced via **Authorization**.
- **Software Application Context:** Implementing rules that dictate which user roles (e.g., Administrator, Basic User) can read specific log files or database tables.
- **Use Cases/Considerations:**
    - **Use When:** Segregating data based on sensitivity or user role (e.g., Need-to-Know).
    - **Not Use When:** Overly complex Access Control Lists (ACLs) which can become unmanageable and lead to configuration errors.

    - **Pros:** Enforces the principle of **Least Privilege** and protects proprietary or sensitive data.
    - **Cons:** Improper configuration can lead to insecure access or deny legitimate users access.

1.1.2 Integrity (I)

Definition: Ensuring that data is accurate, consistent, and complete, and protected from unauthorized or unintentional modification.

1.1.2.1 Hashing

- **Core Concept:** A one-way function that takes input data and produces a fixed-size, unique digital fingerprint (hash value). Any change to the input data results in a completely different hash.
- **Software Application Context:** Storing user passwords (storing the hash, not the plaintext) or verifying the integrity of downloaded files.
- **Use Cases/Considerations:**
    - **Use When:** Verifying data consistency or ensuring that a file has not been tampered with in transit.
    - **Not Use When:** For confidentiality, as hashing is one-way (you can't decrypt it).
    - **Pros:** Highly efficient way to prove data has not changed; essential for password storage.
    - **Cons:** Susceptible to collision attacks if using an outdated algorithm (e.g., MD5).

1.1.2.2 Digital Signatures

- **Core Concept:** A cryptographic technique that uses asymmetric keys (private/public) to prove the authenticity (source) and integrity (unmodified content) of a digital message or file.
- **Software Application Context:** Used in **Code Signing** to verify a software package came from a trusted vendor. Also used for **Nonrepudiation**.
- **Use Cases/Considerations:**
    - **Use When:** Establishing a chain of trust for software components (Supply Chain) or legally binding electronic documents.
    - **Pros:** Provides both integrity and nonrepudiation (the signer cannot deny the action).
    - **Cons:** Requires secure management and protection of the private signing key.

1.1.3 Availability (A)

Definition: Ensuring the system and information are accessible to authorized users when needed.

1.1.3.1 Redundancy & Replication

- **Core Concept:** Duplicating critical hardware, software components, or data to prevent a single point of failure (SPOF). Replication copies data across multiple sites or devices.
- **Software Application Context:** Using mirrored servers or load balancing traffic across multiple identical application instances (**clustering**).
- **Use Cases/Considerations:**
    - **Use When:** Implementing high-availability features for mission-critical services.
    - **Cons:** Higher infrastructure and maintenance costs due to duplication.
    - **Pros:** Ensures continuous operation and quick **failover** in case of a component failure.

1.1.3.2 Resiliency

- **Core Concept:** The system's ability to withstand and recover quickly from failures, errors, or security incidents while maintaining an acceptable level of service.
- **Software Application Context:** Designing APIs with rate limiting to prevent Denial-of-Service (DoS) attacks or building applications to quickly restart upon crash.
- **Use Cases/Considerations:**
    - **Use When:** Designing any public-facing or core business service that must withstand unexpected loads or malicious attacks.
    - **Pros:** Improves user experience during high-stress situations and minimizes data loss.
    - **Cons:** Requires extra development effort to implement robust error handling and self-healing mechanisms.

1.1.4 Authentication (AAA)

Definition: The process of verifying the claimed identity of a user, process, or system (e.g., identity & access management (IAM)).

- **Software Application Context (MFA, SSO, Federated Identity):**
  - **Multi-Factor Authentication (MFA):** Requires a user to provide two or more distinct proofs of identity.
  - **Single Sign-On (SSO):** Allows a user to log in once to gain access to multiple independent systems.
  - **Federated Identity:** Allows a user to use credentials from one service to authenticate to a different, independent service.
- **Use Cases/Considerations:**
  - **Use When:** Accessing any resource with sensitive data or administrative capabilities. MFA is a best practice standard for all access.
  - **Pros:** MFA dramatically reduces the risk of credential theft; SSO improves **Psychological Acceptability** and user convenience.
  - **Cons:** SSO creates a single, high-value target for attackers.

1.1.5 Authorization (AAA)

Definition: The act of granting or denying an authenticated entity the specific permissions or access rights needed to perform an action.

- **Software Application Context (Access Controls, Permissions, Entitlements):** Enforcing **Role-Based Access Control (RBAC)** where permissions are tied to a role (e.g., "Auditor") rather than an individual user.
- **Use Cases/Considerations:**
  - **Use When:** Controlling access granularity within the application (e.g., one user can view data, another can modify it).
  - **Pros:** Enforces **Least Privilege** and limits the scope of damage from a compromised user account.
  - **Cons:** Complex permission models can be difficult to manage, test, and audit.

1.1.6 Accountability (AAA)

Definition: The ability to trace actions and events back to a specific, unique entity (e.g., auditing, logging).

- **Software Application Context:** Storing time-stamped, tamper-proof logs of security-relevant events (e.g., successful/failed logins, configuration changes, critical data access).
- **Use Cases/Considerations:**
    - **Use When:** Any operation that involves sensitive data, configuration, or business-critical logic.
    - **Pros:** Essential for **forensic analysis** and meeting compliance/regulatory requirements.
    - **Cons:** Logging too much data (e.g., sensitive PII, passwords) without masking or encryption can violate confidentiality and privacy requirements.

1.1.7 Nonrepudiation

Definition: Provides undeniable proof of the origin (integrity and authenticity) of data or an action, preventing an entity from falsely denying they performed the action.

- **Software Application Context (Digital Signatures, Block Chain):** Using a user's private key to **digitally sign** a transaction or document.
- **Use Cases/Considerations:**
    - **Use When:** Transactions with high legal or financial consequence (e.g., contract acceptance, financial transfers).
    - **Pros:** Provides a strong, legally defensible mechanism to establish proof of action.
    - **Cons:** The security rests entirely on the protection of the private key used for signing.

1.1.8 Governance, Risk, and Compliance (GRC) Standards

Definition: The structured framework for overseeing security policies, managing security-related risks, and ensuring adherence to mandatory standards (e.g., regulatory authority, legal, industry).

- **Software Application Context:** Building security requirements directly into the SDLC based on regulations like **GDPR** or industry standards like **PCI DSS**.
- **Use Cases/Considerations:**
  - **Use When:** Required for operating in a specific jurisdiction or industry; necessary for defining the security strategy and resource allocation.
  - **Pros:** Provides a clear strategy for security investment and prioritization; avoids legal penalties.
  - **Cons:** Can create organizational friction if GRC policies are not integrated ("shifted left") into the development process early on.

1.2 Understand Security Design Principles

These are high-level rules intended to maximize security throughout the development process.

1.2.1 Least Privilege (PoLP)

- **Definition:** Granting a subject (user, process, or application) only the minimum necessary rights and resources to perform its function. This is fundamental to **Zero Trust**.
- **Software Application Context:** Running a web server process with a non-root, restricted user account; giving a microservice **read-only** database permissions when it doesn't need to write.
- **Use Cases/Considerations:**
  - **Use When:** Mandatory for provisioning service accounts, users, and processes.
  - **Pros:** Drastically limits the impact (**blast radius**) of a successful exploit.
  - **Cons:** Requires more effort during provisioning and maintenance to continually adjust permissions.

1.2.2 Segregation of Duties (SoD)

- **Definition:** Dividing a critical or high-risk task into multiple parts, requiring different individuals or systems to complete them.
- **Software Application Context:** Requiring a developer and a separate security manager to both approve a critical configuration change (**Multi-Party Control**).
- **Use Cases/Considerations:**

    - **Use When:** Managing financial transactions, critical infrastructure changes, or access to master cryptographic keys.
    - **Pros:** Prevents fraud, error, or malicious action by a single person.
    - **Cons:** Can slow down critical business processes; can be bypassed if the two parties collude (**Split Knowledge**).

1.2.3 Defense in Depth

- **Definition:** Employing multiple, layered, and independent security controls to protect resources, so that if one control fails, others are still in place to mitigate the threat.
- **Software Application Context:** Using a **Web Application Firewall (WAF)** at the perimeter, **input validation** in the application code, and **database encryption** (layered controls).
- **Use Cases/Considerations:**
    - **Use When:** Protecting any sensitive application or data (the industry standard).
    - **Pros:** Provides strong overall protection; highly resilient to attacks that bypass a single control.
    - **Cons:** Can be costly and complex to manage and maintain all the different layers.

1.2.4 Resiliency (Fail Safe, Fail Secure)

- **Definition:** The system's ability to maintain an acceptable level of service during a failure, or the principle that, upon failure, the system defaults to a secure (access-denied) state.
- **Software Application Context:** If an authentication service fails, the application **fails secure** by denying all subsequent logins rather than serving them with temporary, potentially insecure, default access.
- **Use Cases/Considerations:**
    - **Use When:** Designing system initialization, error handling, and recovery processes.
    - **Pros:** Prevents security breaches during unforeseen failures; preserves confidentiality and integrity.
    - **Cons:** The *fail safe* state may temporarily violate the **Availability** goal by denying legitimate access.

1.2.5 Economy of Mechanism

- **Definition:** Keeping the security mechanism and its implementation as simple, small, and streamlined as possible.

- **Software Application Context:** Using a widely known and simple **Single Sign-On (SSO)** protocol rather than a complex, proprietary mechanism for centralized authentication.
- **Use Cases/Considerations:**
    - **Use When:** Designing cryptographic modules, authentication flows, or access control checks.
    - **Pros:** Simple systems are easier to review, test, audit, and prove correct (fewer bugs).
    - **Cons:** Avoids necessary complexity in truly advanced security scenarios.

1.2.6 Complete Mediation

- **Definition:** Requiring that all access requests for all resources be checked against the access control policy **every time** the resource is requested, without relying on cached permission decisions.
- **Software Application Context:** A server must **re-verify a session token's validity** with every single request, even if a recent request from the same user succeeded (session management).
- **Use Cases/Considerations:**
    - **Use When:** High-security applications where user roles/permissions may change frequently during a session.
    - **Pros:** Highly effective against permission escalation and session hijacking.
    - **Cons:** Significant performance impact if mediation checks are complex or slow.

1.2.7 Open Design

- **Definition:** The security of a mechanism should not depend on the secrecy of its design or implementation; it should be secure even if an attacker knows all the details (**Kerckhoffs's principle**).
- **Software Application Context:** Using standard, open-source, well-vetted cryptographic libraries for encryption, allowing for **peer review**.
- **Use Cases/Considerations:**
    - **Use When:** Designing cryptographic primitives or security protocols.
    - **Pros:** Allows for widespread community vetting (peer review) and strengthens the system through collaborative testing.
    - **Cons:** Does not protect against vulnerabilities in the implementation, only flaws in the design.

1.2.8 Least Common Mechanism

- **Definition:** Minimizing the amount of security functionality shared by different users or processes to prevent a security flaw in one component from compromising others.
- **Software Application Context:** Deploying different microservices in separate, isolated **containers** or virtual machines to limit the impact of a breach (**compartmentalization/isolation**).
- **Use Cases/Considerations:**
    - **Use When:** Architecting complex, multi-tenant, or microservice applications.
    - **Pros:** Reduces the **blast radius** of an attack; enhances scalability and reliability.
    - **Cons:** Adds system overhead and management complexity due to increased component isolation.

1.2.9 Psychological Acceptability

- **Definition:** Security mechanisms should be designed to be easy, intuitive, and non-intrusive so users accept and use them, rather than finding workarounds.
- **Software Application Context:** Implementing **passwordless authentication** or providing clear, simple screen layouts.
- **Use Cases/Considerations:**
    - **Use When:** Designing any user-facing security interface or workflow (e.g., login, recovery, consent forms).
    - **Pros:** Increases user compliance, reduces security help desk requests
    - **Cons:** Can lead to a compromise in security strength for the sake of convenience (e.g., overly simplified MFA).

1.2.10 Component Reuse

- **Definition:** Utilizing established, pre-vetted, and known secure software components, common controls, or
- **Software Application Context:** Using an organization's pre-approved, certified library for all cryptographic operations (e.g., logging, encryption).
- **Use Cases/Considerations:**
    - **Use When:** Implementing standard security tasks that are not unique to the business logic.
    - **Pros:** Speeds development; relies on code that has already been tested
    - **Cons:** Introduces third-party risk; requires robust **Software Composition Analysis (SCA)** to manage inherited vulnerabilities.

Domain 2 – Secure Software Lifecycle Management – 11%

Domain 2.1 - Manage Security within a Software Development Methodology

Definition: Incorporating security activities (e.g., threat modeling, static analysis, security testing) into the chosen development framework to ensure security is addressed consistently throughout the entire Software Development Life Cycle (SDLC).

2.1.1 Waterfall Methodology

Description

The **Waterfall model** is a sequential, linear approach to software development where progress flows steadily downwards through distinct, successive phases: requirements, design, implementation, testing, deployment, and maintenance. It is characterized by strict documentation and sign-offs; moving to the next phase is only possible after the current one is entirely complete. This model prioritizes predictability and control over flexibility.

- **Security Application Context:** Security activities are mapped directly to their corresponding phase. **Security Requirements** are defined in the requirements phase. A single, comprehensive **Penetration Test** often occurs late in the testing phase, just before release.
- **Use Cases/Considerations:**
  - **Use When:** Requirements are extremely stable, clearly documented, and not expected to change (e.g., small, regulatory-driven projects, or military/safety-critical systems).
  - **Not Use When:** The project is large, complex, or requires continuous user feedback and rapid market response.
  - **Pros:** Easy to manage, highly structured, and strong emphasis on **upfront security documentation and design reviews**.
  - **Cons: Security is often "bolted on"** late, meaning flaws are found only during the testing phase, making them expensive and time-consuming to fix and requiring significant rework.

2.1.2 Agile Methodology

Description

Agile is an iterative and incremental approach that focuses on delivering working software frequently (in short cycles called *sprints*), adapting quickly to change, and emphasizing customer collaboration. It favors flexibility and rapid deployment over rigid, comprehensive upfront documentation. Frameworks like Scrum and Kanban are widely used implementations of Agile.

- **Security Application Context:** Security practices must be integrated into every sprint (the **Shift Left** approach). This involves writing **security stories** (non-functional requirements) into the backlog, performing **micro-threat modeling** per feature, and running automated **Static Application Security Testing (SAST)** with every code build.
- **Use Cases/Considerations:**
    - **Use When:** Requirements are volatile, market needs change rapidly, and continuous feedback is necessary (e.g., web applications, mobile apps).
    - **Not Use When:** Strict, comprehensive upfront control and extensive regulatory documentation are required before implementation can start.
    - **Pros: Security flaws are found and fixed early** due to frequent testing and small changes; security becomes a continuous activity, reducing expensive rework.
    - **Cons:** The rapid pace can lead teams to skip manual security steps (like detailed code review or full risk analysis) if they are not automated, potentially accruing **security debt**.

2.1.3 DevSecOps Methodology

Description

DevSecOps extends the DevOps philosophy by integrating security into every part of the continuous pipeline, from initial code commit to production monitoring. It champions **automation** and collaboration, making security a shared responsibility across development, security, and operations teams. The core principle is "Security as Code."

- **Security Application Context:** Security is **fully automated** within the CI/CD pipeline. This includes running tools like **SAST**, **DAST**, and **Software Composition Analysis (SCA)** automatically on every code change. **Infrastructure as Code (IaC)** is used to provision secure baseline configurations in deployment environments.
- **Use Cases/Considerations:**

    - **Use When:** High deployment frequency is required, and the infrastructure is highly automated (e.g., cloud-native applications, microservices).
    - **Not Use When:** The organization lacks the maturity, automation tools, or skill set to manage and secure complex, automated pipelines.
    - **Pros: Fastest remediation cycle** and **highest scalability** for security checks. Security is proactive, continuous, and integrated into developer workflows.
    - **Cons:** High initial investment in tools and expertise; a security vulnerability in the pipeline tools or configuration itself can compromise the entire deployment chain.

2.1.4 Spiral Model

Description

The **Spiral model** is a risk-driven methodology that combines the sequential nature of Waterfall with the iterative aspect of prototyping. It is structured around cycles of four activities: **determination of objectives**, **risk assessment and reduction**, **engineering/development**, and **planning the next iteration**. The project scope grows incrementally as it moves through successive spirals.

- **Security Application Context:** Security is inherently addressed through the mandatory **risk assessment phase** in every spiral. This formal, recurring assessment ensures that the security implications of new features or architecture changes are constantly re-evaluated.
- **Use Cases/Considerations:**
    - **Use When:** Large, expensive, and high-risk projects where management needs constant, formal visibility into risk (e.g., new product development where scope is not fully defined).
    - **Pros:** Excellent for managing and **mitigating risk incrementally**; adapts well to evolving requirements because risk drives development priorities.
    - **Cons:** More complex and resource-intensive than simple Waterfall or Scrum; the constant risk assessment can be time-consuming if not scoped efficiently.

2.1.5 Extreme Programming (XP)

Description

Extreme Programming (XP) is a specific, disciplined subset of the Agile methodology. It is characterized by core technical practices such as very short development cycles, frequent releases, heavy reliance on **automated testing**, **pair programming**, and **test-driven development (TDD)**. It emphasizes simplicity, communication, and immediate feedback.

- **Security Application Context:** TDD can be extended to **Security Test-Driven Development (Security TDD)**, where security tests (based on abuse/misuse cases) are written *before* the code is implemented. **Pair programming** serves as a continuous, informal **peer code review** for security.
    - **Use When:** Small- to medium-sized, co-located teams on fast-moving, technically demanding projects where code quality is a high priority.
    - **Pros:** Results in high quality, well-tested code; provides a **built-in, continuous security review** mechanism via pair programming.
    - **Cons:** Requires a highly disciplined and collaborative team; practices like pair programming are not always feasible or productive for all team members or distributed teams.

Domain 2.2 - Identify and Adopt Security Standards (11%) - Comprehensive Guide

Definition: Establishing and implementing a formal set of security policies, models, and training programs across the organization to standardize secure development practices, often guided by external compliance mandates or best practices.

2.2.1 Categorization of Security Standards

Security standards and guidance can be broadly classified based on their source and mandatory nature:

2.2.1.1 External Standards (Mandatory/Voluntary)

These are established outside the organization by governments, international bodies, or industry groups.

- **Regulatory/Legal:** Mandatory laws established by government entities (e.g., GDPR, HIPAA).
- **Industry/Contractual:** Standards required for conducting business in a specific sector (e.g., PCI DSS) or mandated by a client contract.
- **Voluntary/Best Practice:** Guidance offered for continuous improvement (e.g., ISO, NIST, OWASP).

2.2.1.2 Internal Standards

These are established and enforced *within* the organization to provide specific, actionable rules derived from external mandates or organizational risk tolerance.

- **Policies:** High-level, non-negotiable rules set by management (e.g., "All code must be scanned before release").
- **Standards/Guidelines:** Specific technical rules (e.g., "All passwords must be hashed using PBKDF2").
- **Secure Coding Guidelines:** Detailed, language-specific rules for developers (e.g., "Never concatenate user input directly into a SQL query").

2.2.2 External Regulatory and Industry Standards

2.2.2.1 Payment Card Industry Data Security Standard (PCI DSS)

- **Core Concept:** A proprietary information security standard administered by the PCI Security Standards Council. It applies to all entities that store, process, or transmit cardholder data (CHD). Compliance is **mandatory** for any business handling credit card transactions.
- **Software Application Context:** PCI DSS Requirement 6 specifically mandates secure software development practices, including formal change control, testing, and training (secure coding).
- **Use Cases/Considerations:**
  - **Use When:** Processing credit card payments.
  - **Pros:** Reduces risk of financial fraud; required for business operations.
  - **Cons:** Very strict and often requires extensive, costly annual audits.

2.2.2.2 Federal Information Processing Standards (FIPS)

- **Core Concept:** Standards and guidelines developed by NIST for U.S. federal government computer systems (excluding national security systems). They are **mandatory** for federal agencies and contractors. **FIPS 140-3** is the standard for cryptographic modules.
- **Software Application Context:** Any software component (e.g., an encryption library) used by a federal agency must use cryptography that has been **FIPS 140-3 validated**.
- **Use Cases/Considerations:**
  - **Use When:** Developing software for U.S. government contracts.
  - **Pros:** Ensures a high, standardized level of cryptographic security.
  - **Cons:** Validation is a lengthy and expensive process.

2.2.3 External Best Practice Frameworks

2.2.3.1 National Institute of Standards and Technology (NIST)

- **Core Concept:** Provides non-regulatory but widely adopted guidance. Key to the SDLC is the **Risk Management Framework (RMF)** and controls from **SP 800-53**.
- **Flow/Structure: The Six-Step RMF:**
    1. **Categorize:** Define the system's impact level (CIA).
    2. **Select:** Choose baseline security controls (SP 800-53) based on categorization.
    3. **Implement:** Install and document the selected controls.
    4. **Assess:** Evaluate control effectiveness.
    5. **Authorize (ATO):** Senior official accepts the risk and grants permission to operate.
    6. **Monitor:** Continuously track the system and its controls.
- **Software Application Context:** Controls from **SP 800-53** (e.g., AC-4 Information Flow Enforcement) are implemented during the design phase.
- **Use Cases/Considerations:**
    - **Use When:** Seeking highly detailed, publicly available, prescriptive guidance.
    - **Pros:** Provides a clear, repeatable security lifecycle; **detailed guidance** for implementation.
    - **Cons:** The sheer volume of documentation can be overwhelming.

2.2.3.2 Software Assurance Forum for Excellence in Code (SAFECode)

- **Core Concept:** An organization that produces free, practical publications on software assurance, particularly focusing on **security training, secure engineering, and supply chain integrity**. It is driven by leading software and hardware companies.
- **Software Application Context:** Developers use SAFECode guides to understand how to implement **secure engineering practices** within their development pipeline, especially around third-party components.
- **Use Cases/Considerations:**
    - **Use When:** Establishing a comprehensive secure development training and process program based on industry consensus.
    - **Pros:** Highly practical and actionable guidance from top industry experts.
    - **Cons:** Not a formal certification or compliance standard.

2.2.3.3 Information Technology Infrastructure Library (ITIL)

- **Core Concept:** A set of detailed practices for **IT Service Management (ITSM)**, focusing on aligning IT services with the needs of the business. While not a security standard, it defines the essential service management processes (e.g., change management, incident management) that secure software must operate within.
- **Software Application Context:** A newly deployed application must integrate its security logging capabilities with the organization's existing ITIL-governed **Incident Management** process.
- **Use Cases/Considerations:**
  - **Use When:** Integrating secure software into a large, managed enterprise IT environment.
  - **Pros:** Ensures consistent, high-quality **Service, Change, and Release Management**, which are critical for security operations.
  - **Cons:** Focuses on the *process* of service delivery, not the *security* of the code itself.

2.2.4 International Standardization Organization (ISO) Frameworks

2.2.4.1 ISO/IEC 27001 / 27002 (ISMS)

- **Core Concept: ISO 27001** sets the requirements for an **Information Security Management System (ISMS)**, requiring an organization to manage risk to its information assets, including software.
- **Software Application Context:** Requires implementing controls from **ISO 27002** related to development and acquisition, such as secure development policies and supplier relationship management.
- **Use Cases/Considerations:**
  - **Use When:** Requiring global recognition for security management (formal certification).
  - **Pros:** Provides a **holistic, internationally recognized framework** based on the PDCA cycle.
  - **Cons:** Certification is **resource-intensive** and high-level, requiring significant local translation.

2.2.4.2 ISO/IEC 15408 and Common Criteria (CC)

- **Core Concept: ISO 15408** provides the standardized methodology and requirements for **evaluating and certifying the security of IT products**. The framework is known as the **Common Criteria (CC)**.
- **Flow/Structure:** Products are evaluated against a **Security Target (ST)** and a **Protection Profile (PP)** to achieve an **Evaluation Assurance Level (EAL)** from 1 to 7.
- **Software Application Context:** Used when procuring commercial security products (e.g., a firewall, OS, or cryptographic module) where high assurance of security function is required.
- **Use Cases/Considerations:**
    - **Use When:** Developing or procuring high-assurance security products, often for government/defense use.
    - **Pros:** Provides a **globally recognized, repeatable, and thorough evaluation** of security features.
    - **Cons:** Evaluation is extremely complex, expensive, and time-consuming.

2.2.4.3 ISO/IEC 21827:2008 (System Security Engineering Capability Maturity Model - SSE-CMM)

- **Core Concept:** A model designed to assess and rate the **security engineering maturity** of an organization or project, allowing for continuous improvement of security practices. It focuses on the security processes used during development and engineering.
- **Software Application Context:** Used by an organization to determine its security capability level (similar to CMMI) and define a strategy to achieve higher maturity in its secure development process.
- **Use Cases/Considerations:**
    - **Use When:** Assessing the maturity of a security engineering team or a specific high-risk development project.
    - **Pros:** Provides a structured way to **measure security process quality** and define improvement roadmaps.
    - **Cons:** Less commonly adopted than models like SAMM or BSIMM.

2.2.4.4 ISO/IEC 25000 (SQuaRE)

- **Core Concept:** A series of standards (Software Quality Requirements and Evaluation) focusing on **software quality**. The security aspects fall under the **functional suitability and reliability characteristics** of quality.
- **Software Application Context:** Used to ensure the delivered software meets defined quality standards, including security features.

- **Use Cases/Considerations:**
    - **Use When:** Defining quality acceptance criteria for software deliverables.
    - **Pros:** Provides a comprehensive model for defining and evaluating all software quality attributes.

2.2.4.5 ISO 28000

- **Core Concept:** Management system for **supply chain security**. While not strictly software, it is highly relevant to **Domain 8 (Secure Software Supply Chain)** as it governs the overall security of components, processes, and logistics.
- **Software Application Context:** Used to manage the risk associated with hardware or third-party software components received from suppliers.
- **Use Cases/Considerations:**
    - **Use When:** Managing vendors and partners who contribute components to the final product.
    - **Pros:** Provides a structured, auditable approach to supply chain risk.

2.2.5 Maturity and Awareness Models

2.2.5.1 Software Assurance Maturity Model (SAMM)

- **Core Concept:** OWASP framework for measuring and incrementally improving the maturity of a security program across five functions.
- **Software Application Context:** Used to score current secure coding practices on a scale of 0 to 3 to identify weak areas.
- **Use Cases/Considerations:**
    - **Use When:** An organization wants a flexible, measurable roadmap to incrementally improve its security program.
    - **Pros: Customizable** and provides clear steps for **measurable progress**.

2.2.5.2 Building Security in Maturity Model (BSIMM)

- **Core Concept:** A data-driven model that **benchmarks** an organization's Security Assurance Program (SAP) against industry peers across 12 practices.
- **Software Application Context:** A company compares its security testing frequency against the metrics of similar-sized companies.
- **Use Cases/Considerations:**

    - **Use When:** Benchmarking a mature security program against industry leaders.
    - **Pros:** Based on **real-world data** from a large collection of organizations.

2.2.5.3 Promoting Security Awareness

- **Core Concept:** Mandatory, recurring, role-based training programs to ensure all personnel, especially developers, know their security responsibilities.
- **Software Application Context:** Requiring developers to complete training on the **OWASP Top 10** before being granted access to production code repositories.
- **Use Cases/Considerations:**
    - **Use When:** Onboarding new staff or after a significant security event.
    - **Pros:** Directly addresses the root cause of many vulnerabilities (developer error); improves compliance.

Domain 2.3 - Outline Strategy and Roadmap

Definition: Establishing a clear, long-term plan for the Software Security Assurance (SSA) program, defining measurable goals, and implementing mandatory checkpoints within the Software Development Life Cycle (SDLC) to enforce security standards.

2.3.1 Security Strategy and Roadmap

Definition

The **security strategy** defines the high-level, multi-year objectives for the SSA program (e.g., "Achieve SAMM Level 2 in Verification by 2027"). The **roadmap** translates this strategy into actionable projects, timelines, and resource allocation, showing *how* the organization will reach those objectives.

- **Core Concept:** Translating organizational risk tolerance and regulatory mandates into a sequential plan of security activities and investments.
- **Flow/Structure:** The roadmap typically aligns with product release cycles and includes phases like **Assessment** (benchmarking current maturity), **Prioritization** (selecting high-impact security practices), and **Implementation** (rolling out new tools or training).
- **Software Application Context:** The strategy might mandate the integration of a **Static Application Security Testing (SAST)** tool. The roadmap defines that SAST must be integrated into the CI pipeline of the three most critical applications within the next six months.
- **Use Cases/Considerations:**
    - **Use When:** Justifying security budget, planning resource needs, and aligning the security team's goals with the overall business objectives.
    - **Not Use When:** The plan must be rigid; effective roadmaps are **living documents** that must be adjusted based on new risks or project changes.
    - **Pros:** Ensures security efforts are **consistent, prioritized, and measurable** over time; shifts the security program from reactive firefighting to proactive engineering.
    - **Cons:** Requires strong executive buy-in and consistent funding; a poorly executed roadmap can waste resources and damage credibility.

2.3.2 Security Milestones and Checkpoints

Definition

Specific, mandatory points within the SDLC where a project must pause to demonstrate compliance with security requirements. These serve as **security gates** that prevent non-compliant code or design from moving forward.

2.3.2.1 Milestones and Checkpoints

- **Core Concept:** Formal sign-off points where the security team (or designated security champion) reviews evidence that specific security activities have been completed successfully.
- **Flow/Structure:** These are typically placed at the end of key phases, such as **requirements sign-off**, **design freeze**, and **pre-deployment review**.
- **Software Application Context:** A milestone might be defined as: "The Security Architecture Review must be signed off by the lead architect and security team before code implementation begins."
- **Use Cases/Considerations:**
    - **Use When:** High-risk applications or systems subject to strict regulatory compliance (e.g., PCI, FIPS).
    - **Pros: Enforces security policy** at critical junctures; ensures high-risk design flaws are identified before they become costly to fix.
    - **Cons:** If manual and frequent, checkpoints can significantly slow down development velocity, especially in Agile/DevOps environments.

2.3.2.2 Control Gate

- **Core Concept:** A decision point that automatically or manually stops a process until a set of defined criteria is met. This is a mechanism for enforcing a security milestone.
- **Flow/Structure:** A mandatory gate integrated into the CI/CD pipeline.
- **Software Application Context:** A configuration management system is set up to prevent (gate) any deployment artifact from reaching the staging environment unless it has a valid, untampered **Code Signature**.
- **Use Cases/Considerations:**
    - **Use When:** Automating compliance enforcement within the DevOps pipeline (DevSecOps).
    - **Pros: High consistency** and immediate feedback; prevents non-compliant components from moving down the chain.
    - **Cons:** A failure in the gate mechanism (e.g., a buggy SAST tool) can halt the entire pipeline unnecessarily.

2.3.2.3 Break/Build Criteria

- **Core Concept:** Specific, quantifiable rules that determine whether a software build is deemed secure enough to proceed. If the criteria are not met (the build **breaks**), the development process stops.
- **Flow/Structure:** These are integrated directly into the automated build process and version control.
- **Software Application Context: Criteria Example:** "The build must **break** if the DAST scan reports more than zero critical vulnerabilities, or if the Software Composition Analysis (SCA) tool detects any unpatched vulnerability with a **Common Vulnerability Scoring System (CVSS)** score above 8.0 in a third-party library."
- **Use Cases/Considerations:**
  - **Use When:** Defining clear standards of quality and acceptance for automated testing phases.
  - **Pros:** Provides an objective, automated measure of security quality, eliminating human subjectivity in accepting risk.
  - **Cons:** Requires careful tuning; overly strict criteria can lead to high friction if the testing tools produce frequent false positives.

Domain 2.4 - Define and Develop Security Documentation

Definition: The creation and maintenance of formal, auditable records that capture security requirements, design decisions, implementation details, and operational procedures throughout the SDLC. Documentation provides accountability, ensures repeatability, and is mandatory for compliance.

2.4.1 Policy and Governance Documentation

These high-level documents define the rules and constraints governing the application, data, and personnel.

2.4.1.1 Application Security Policy

- **Core Concept:** A formal, high-level document stating the organization's rules and commitment to security for a specific application or application portfolio. It mandates security requirements and practices.
- **Software Application Context:** States that the application must use only approved encryption algorithms (FIPS-validated), and all user input must be validated and sanitized.
- **Use Cases/Considerations:**
  - **Use When:** Establishing the security baseline for the application.
  - **Pros:** Provides clear, mandated security requirements, often derived from GRC standards (Domain 1.1.8).
  - **Cons:** Must be continually updated to avoid conflicts with new architectural decisions.

2.4.1.2 Data Classification Policy Documentation

- **Core Concept:** Defines the categories of data (e.g., Public, Internal, Confidential, Restricted) and the security controls required for each category throughout its lifecycle.
- **Software Application Context:** Defines that **Confidential** data (e.g., PII) must be encrypted at rest and in transit, while **public** data requires only basic access control. This directly informs the security design (Domain 4).
- **Use Cases/Considerations:**
  - **Use When:** Handling multiple types of data with varying legal and business risk levels.
  - **Pros:** Ensures consistent application of security controls based on sensitivity; mandatory for compliance (e.g., GDPR).

2.4.1.3 Data Retention Policy Documentation

- **Core Concept:** Specifies how long different categories of data must be kept before being securely disposed of (destroyed). This is governed by legal, regulatory, and business needs.
- **Software Application Context:** Defines that application logs must be kept for 90 days for forensic purposes, but audit trails for financial transactions must be retained for seven years. This informs the design of the database and archiving systems.
- **Use Cases/Considerations:**
    - **Use When:** Compliance with privacy and finance laws.
    - **Pros:** Prevents unnecessary storage costs and reduces legal exposure from retaining PII beyond the legal limit.

2.4.2 Development and Design Documentation

These documents capture the security architecture, coding standards, and implementation details.

2.4.2.1 Security Architecture Document (SAD)

- **Core Concept:** A formal blueprint detailing how security controls are integrated into the application and the underlying infrastructure (infra). It outlines trust boundaries, data flows, and placement of security components.
- **Software Application Context (Design with Infra):** Shows the architecture of security controls, including the placement of the **Web Application Firewall (WAF)** (infra component) in front of the application servers, the use of a separate **secrets management service** (infra/security component), and the network segmentation (infra) between tiers.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for complex systems, cloud deployments, or **Security Design Reviews** (Domain 4).
    - **Pros:** Essential for threat modeling; ensures stakeholders agree on the security posture *before* coding.

2.4.2.2 Security Implementation Documentation

- **Core Concept:** Detailed records on *how* the approved security design was actually coded and configured. This includes specific configuration parameters for security products and code examples.
- **Software Application Context (Security Product Documentation):** For a **Runtime Application Self-Protection (RASP)** product, this document lists the specific rules

enabled, the deployment mode (monitor vs. block), and the integration points within the application's runtime environment.

- **Use Cases/Considerations:**
    - **Use When:** Auditing or reproducing the production environment configuration.
    - **Pros:** Critical for verifying that the implemented controls match the approved design.

2.4.2.3 Coding Standards and Guidelines Documentation

- **Core Concept:** A set of language-specific or platform-specific rules that developers must follow to ensure security (e.g., "Always use prepared statements to prevent SQL Injection").
- **Software Application Context:** Provides concrete examples of secure vs. insecure code snippets for the application's primary programming language. This is tied directly to **Secure Coding Training** (Domain 2.2).
- **Use Cases/Considerations:**
    - **Use When:** Scaling a development team or mitigating recurring, common code vulnerabilities.
    - **Pros:** Standardizes code quality and reduces the volume of flaws found later by automated testing.

2.4.2.4 Code Changes Documentation

- **Core Concept:** Detailed records within the version control system (VCS) or ticketing system that track and document every modification to the codebase, including the purpose, security review status, and approval.
- **Software Application Context:** A ticket detailing a bug fix for an XSS vulnerability must be linked to a code commit and include evidence of a successful security peer review and **regression test** (Domain 6).
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for all development; essential for forensic review and maintaining integrity.
    - **Pros:** Provides clear **accountability** (Domain 1.1.6) and supports the change management process (Domain 2.9).

2.4.3 Operations, User, and Maintenance Documentation

These documents support the application after deployment.

2.4.3.1 User Guides and Administrator Guides

- **Core Concept:** Documentation tailored to specific user roles to ensure the application's security features are used correctly.
- **Software Application Context:**
    - **User Guides:** Explain how to use the application's **MFA** feature, what data the application collects, and how to exercise **data rights** (privacy, Domain 3).
    - **Administrator Guides:** Detail how to manage user roles, review security logs, and securely install/configure the application.
- **Use Cases/Considerations:**
    - **Use When:** Required for any application with configurable security features.
    - **Pros:** Increases the **Psychological Acceptability** (Domain 1.2.9) of security features; ensures correct operational use.

2.4.3.2 Incident Response (IR) Plan Documentation

- **Core Concept:** A detailed, step-by-step plan for the security team to follow immediately upon the detection of a security breach or incident.
- **Software Application Context:** Specifies how to **contain** a running application (e.g., blocking network traffic), where to find application and security product logs, and who is responsible for forensics and remediation.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for all production systems; part of **secure operation practices** (Domain 2.9).
    - **Pros:** Enables a rapid, organized, and compliant response, minimizing damage.

2.4.3.3 Business Continuity Plan (BCP) and Disaster Recovery (DR) Plan Documentation

- **Core Concept:**
    - **BCP:** High-level plan focused on keeping business functions operational during a major disruption.
    - **DRP:** Technical plan detailing the procedures and resources needed to restore IT infrastructure and systems following a disaster.

- **Software Application Context:** The **DRP** details the **failover** process, the recovery site location, and the restoration order of application services to meet defined **Recovery Time Objectives (RTO)** and **Recovery Point Objectives (RPO)**.
- **Use Cases/Considerations:**
  - **Use When:** Required to meet availability and resiliency goals (Domain 1.1.3 and 1.2.4).
  - **Pros:** Essential for maintaining business operations and integrity during large-scale failures.

2.4.3.4 Logging and Monitoring Documentation

- **Core Concept:** Details what events the application generates (what to log), how logs are protected (log integrity/confidentiality), and how they are monitored (alerts).
- **Software Application Context:** Specifies which security events (e.g., failed authorization attempts, successful privilege escalations) are forwarded to the **Security Information and Event Management (SIEM)** system and the thresholds for generating alerts.
- **Use Cases/Considerations:**
  - **Use When:** Mandatory for forensic readiness and continuous security monitoring (Domain 7.7).
  - **Pros:** Critical for detecting attacks in real-time and providing audit trails (**Accountability**).

2.4.3.5 Security Metrics Documents

- **Core Concept:** Reports and dashboards that capture Key Performance Indicators (KPIs) and Key Risk Indicators (KRIs) to measure the effectiveness of the security program.
- **Software Application Context:** Includes reports on **Average Remediation Time (ART)** for vulnerabilities, the percentage of code covered by SAST, and the trend of open critical vulnerabilities over time.
- **Use Cases/Considerations:**
  - **Use When:** Reporting to executive management, justifying security investments, and driving continuous improvement (Domain 2.5).
  - **Pros:** Allows for data-driven decisions and proves the value of the security program.

Domain 2.5 - Define Security Metrics

Definition: Establishing quantifiable measures (metrics) and organizational goals (objectives) to effectively monitor, measure, and report on the status, effectiveness, and efficiency of the Software Security Assurance (SSA) program and its associated risks.

2.5.1 Goals and Strategy Metrics (KPIs and OKRs)

2.5.1.1 Key Performance Indicators (KPI)

- **Core Concept:** Quantifiable measures used to evaluate the success of an organization or a specific activity in meeting its objectives. KPIs are used to track overall program health and efficiency.
- **Flow/Structure:** KPIs are typically historical and focus on **performance**. Examples include: "Percentage of critical applications covered by annual penetration testing" or "Time spent on manual security reviews."
- **Software Application Context:** A KPI might be set as: "Maintain a vulnerability density of less than 0.5 High-severity findings per 1,000 lines of code."
- **Use Cases/Considerations:**
    - **Use When:** Reporting to senior management on the program's overall health and historical trends.
    - **Pros:** Provides a clear, high-level view of program status; essential for justifying budgets and resource allocation.
    - **Cons:** Focusing on only a few KPIs may obscure deeper issues.

2.5.1.2 Objectives and Key Results (OKR)

- **Core Concept:** A goal-setting framework that defines a bold **Objective** (what is to be achieved) and **Key Results** (how success will be measured, usually 3-5 measurable metrics). OKRs are aspirational and future-focused.
- **Flow/Structure: Objective:** "Shift security left to fully integrate into the CI/CD pipeline." **Key Result:** "Achieve 95% automated SAST scanning coverage for all new code by Q4."
- **Software Application Context:** An application team sets an objective to "Eliminate all external audit findings." A key result is: "Achieve a zero-finding result on the next pre-deployment security assessment."
- **Use Cases/Considerations:**
    - **Use When:** Driving change and improvement (e.g., moving from a Waterfall security process to DevSecOps).
    - **Pros:** Aligns security efforts across different teams (Dev, Sec, Ops); results are clearly measurable.

    - **Cons:** Setting weak or non-ambitious goals is counterproductive.

2.5.2 Risk and Prioritization Metrics

2.5.2.1 Criticality Level (Risk Scoring)

- **Core Concept:** A rating that assesses the severity of a vulnerability based on its potential impact to the organization's CIA (Confidentiality, Integrity, Availability) triad.
- **Flow/Structure:** Often defined using the **Common Vulnerability Scoring System (CVSS)**, which combines **Base Metrics** (e.g., Attack Vector, Impact) with **Temporal** and **Environmental Metrics** to produce a final score (Low, Medium, High, Critical).
- **Software Application Context:** A flaw allowing unauthorized access to a **Confidential** (high criticality) database is assigned a CVSS score of 9.8 (Critical), mandating immediate remediation.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for prioritizing vulnerability remediation. Used to establish **break/build criteria** (Domain 2.3).
    - **Pros:** Provides an objective, standardized method for comparing and prioritizing flaws across different systems.
    - **Cons:** Technical CVSS scores must be combined with business context (e.g., asset value) to accurately represent **business risk**.

2.5.2.2 Complexity

- **Core Concept:** A metric used to assess the effort required to fix a vulnerability, often measured by the inherent intricacy of the affected code or the number of components requiring change.
- **Flow/Structure:** Complexity is measured during the triage phase. A fix requiring a major overhaul of a legacy authentication module is assigned **High** complexity, which affects the ART goal.
- **Software Application Context:** Used in risk analysis to help balance technical risk against the actual **cost and time** of fixing the flaw.
- **Use Cases/Considerations:**
    - **Use When:** Factoring remediation time into the overall project planning (Risk Analysis, Domain 2.8).
    - **Pros:** Helps manage developer workload and avoid unrealistic deadlines.
    - **Cons:** Highly subjective and relies on developer/architect input, requiring standardization.

2.5.3 Program Effectiveness Metrics (Flow Metrics)

2.5.3.1 Average Remediation Time (ART)

- **Core Concept:** The average time elapsed from when a vulnerability is first discovered (or reported) to when the fix is deployed to the production environment. This measures the efficiency of the security pipeline.
- **Flow/Structure:** ART is typically tracked by criticality level. For example, "ART for Critical findings is 5 days." The counter stops when the deployment pipeline confirms the patched code is live.
- **Software Application Context:** Used to monitor the effectiveness of the **Incident Response Plan** (Domain 2.9) and vulnerability management process (Domain 7.10).
- **Use Cases/Considerations:**
    - **Use When:** Measuring the organization's speed of response to risk and minimizing exposure time.
    - **Pros:** A clear, objective measure of security agility.
    - **Cons:** Can be manipulated if teams delay reporting discovery.

2.5.3.2 ASVS Compliance (Verification Coverage Metric)

- **Core Concept:** Using the **Application Security Verification Standard (ASVS)** as a **coverage metric** to determine the completeness and depth of security testing and compliance for a given assurance level.
- **Flow/Structure:** The metric is expressed as a percentage: $(\text{Number of ASVS requirements met}) / (\text{Total ASVS requirements for target level})$. For a business-critical application targeting **ASVS Level 2 (L2)**, the team tracks "L2 Compliance: 85%."
- **Software Application Context:** If the organization mandates 100% L2 compliance for all production apps, the ASVS compliance score is used as a **Go/No-Go Gate** for deployment (Domain 2.3).
- **Use Cases/Considerations:**
    - **Use When:** Objectively measuring the completeness of security testing (Domain 6) and requirements implementation (Domain 3).

- **Pros:** Provides a non-subjective, internationally recognized measure of assurance. Allows auditors to quickly verify the security posture.
- **Cons:** Requires significant upfront effort to map requirements to test cases.

2.5.3.3 Other Key Effectiveness Metrics

- **Vulnerability Density:** The number of vulnerabilities per 1,000 lines of code (LOC). Used to compare the security quality of different applications or teams.
- **False Positive Rate:** The percentage of issues flagged by automated tools (SAST/DAST) that are not, in fact, real vulnerabilities. High rates lead to **alert fatigue**.
- **Security Debt:** The cumulative measure of all outstanding vulnerabilities (often weighted by CVSS score) that the organization has chosen to accept or defer. Used for risk tracking (Domain 2.8).
- **Testing Coverage:** The percentage of code (or security requirements) covered by automated security scans or tests. This directly measures the completeness of the **Verification** phase (Domain 6).

Domain 2.6 - Decommission Applications

Definition: The structured, formal, and secure process of retiring a software application, service, or system component, ensuring all associated assets—including data, credentials, licenses, and documentation—are managed or destroyed in a policy-compliant manner.

2.6.1 End of Life (EOL) Policies and System Management

These policies focus on formally retiring the *application environment* and its components, ensuring no abandoned attack surfaces remain.

2.6.1.1 Credential Removal

- **Core Concept:** The mandatory process of identifying and revoking all user accounts (including service accounts, administrator accounts, and API keys) that were provisioned specifically for the retiring application.
- **Flow/Structure:** Must follow the principle of **Least Privilege** and **Accountability**. An audit trail of all removed accounts is required. This often involves checking IAM systems and secrets managers.
- **Software Application Context:** Deleting the application's dedicated database service account, revoking its cloud API key, and removing the developer group's access to its version control repository.
- **Use Cases/Considerations:**

    - **Use When:** Any application is permanently removed from the production or testing environment.
    - **Pros:** Eliminates potential **"zombie accounts"** that could be exploited by an attacker for unauthorized access (Lateral Movement).
    - **Cons:** Must be performed carefully to avoid breaking dependencies with other systems that might have accidentally shared these credentials.

2.6.1.2 Configuration Removal

- **Core Concept:** The process of securely wiping or disabling all production configurations, network entries, firewall rules, and virtual machine images associated with the application.
- **Flow/Structure:** All configurations defined via **Infrastructure as Code (IaC)** should be destroyed using the code, rather than manual deletion, to ensure completeness and verification.
- **Software Application Context:** Removing firewall exceptions, deleting load balancer entries, terminating cloud instances, and removing registry entries or daemon configurations.
- **Use Cases/Considerations:**
    - **Use When:** Removing any application environment component.
    - **Pros:** Reduces the attack surface by eliminating unnecessary active components and adhering to the **Least Common Mechanism** principle.
    - **Cons:** Failure to remove all network configurations (e.g., DNS entries) can lead to **subdomain takeover** or traffic routing errors.

2.6.1.3 License Cancellation and Service-Level Agreements (SLA)

- **Core Concept:** The formal termination of all commercial contracts and legal agreements related to the decommissioned application (e.g., third-party library licenses, vendor support, hosting agreements).
- **Flow/Structure:** Requires coordination between security, IT finance, and legal teams to ensure all contractual obligations are met (e.g., minimum retention periods) before cancellation.
- **Software Application Context:** Cancelling the annual license for a **Web Application Firewall (WAF)** or a database product that was exclusively used by the retiring application.
- **Use Cases/Considerations:**
    - **Use When:** Shutting down an application to avoid recurring costs.

    - **Pros:** Avoids unnecessary costs and prevents compliance violations related to licensing usage.

2.6.1.4 Archiving

- **Core Concept:** The process of creating and securing a final, immutable copy of the application's source code, logs, audit trails, and security documentation for legal or historical purposes.
- **Flow/Structure:** The archived data must be encrypted and stored in a secure, segmented repository with strict access controls to meet **Confidentiality** and **Integrity** requirements.
- **Software Application Context:** Creating a final, signed copy of the source code repository and the last 90 days of security logs, encrypting them with a separate, long-term key, and transferring them to a secure offline storage system.
- **Use Cases/Considerations:**
    - **Use When:** Compliance dictates a history of the software must be retained, or for potential future litigation/forensics.
    - **Cons:** Archiving logs that contain sensitive PII/secrets can create a new long-term risk if not properly encrypted and managed.

2.6.2 Data Disposition

Data disposition focuses specifically on handling the sensitive data created, stored, or processed by the application.

2.6.2.1 Retention

- **Core Concept:** The mandatory compliance step of retaining application data for a specific legal, regulatory, or business period before destruction. This is governed by the **Data Retention Policy** (Domain 2.4).
- **Flow/Structure:** Data is moved from active storage to secure, restricted, and often tiered archival storage for the mandated period.
- **Software Application Context:** Financial data may be moved to long-term storage for seven years to comply with tax laws. PII may need to be anonymized or destroyed immediately upon user request or project completion.
- **Use Cases/Considerations:**
    - **Use When:** Processing sensitive data with clear regulatory timelines (e.g., GDPR, HIPAA).
    - **Pros:** Ensures regulatory compliance and legal defensibility.

    - **Cons:** Retaining data longer than necessary increases the organization's **attack surface** and liability.

2.6.2.2 Destruction

- **Core Concept:** The process of permanently and irreversibly sanitizing storage media and data such that it cannot be recovered by any known means.
- **Flow/Structure:** Methods include **degaussing** (for magnetic media), **physical destruction** (e.g., shredding), or **cryptographic erasure** (destroying the encryption key rather than the data itself, which is faster). Destruction must be logged and verified.
- **Software Application Context:** Deleting the application's database schema and then using a certified overwriting utility on the underlying storage media, or simply deleting the master encryption key used for the database volume (cryptographic erasure).
- **Use Cases/Considerations:**
    - **Use When:** The data has passed its retention period and is no longer needed.
    - **Pros:** Mandatory for privacy compliance (e.g., Right to be Forgotten); minimizes risk of data leakage from decommissioned assets.
    - **Cons:** Must be carefully executed and verified; an incomplete wipe violates **Integrity** and **Confidentiality**.

2.6.2.3 Dependencies

- **Core Concept:** The step of identifying all upstream (data provider) and downstream (data consumer) systems that rely on the retiring application for data, services, or functionality.
- **Flow/Structure:** Requires a comprehensive system inventory review. Dependent systems must either be migrated to a new data source, decommissioned, or updated to tolerate the loss of the retiring system (e.g., using a **Fail Safe** approach).
- **Software Application Context:** Before decommissioning a legacy customer data service, the process must confirm that no active billing or reporting services are querying it.
- **Use Cases/Considerations:**
    - **Use When:** Any shared service or critical data repository is being retired.
    - **Pros:** Prevents cascading failures that could impact the **Availability** of other mission-critical systems.
    - **Cons:** Dependency mapping can be complex and is often overlooked, leading to unexpected service interruptions post-decommissioning.

Domain 2.7 - Create Security Reporting Mechanisms

Definition: Establishing formal systems and processes to consistently gather, analyze, and present security-related data to relevant stakeholders (developers, management, and executive leadership). Effective reporting drives decision-making, prioritizes risk, and fosters accountability.

2.7.1 Reporting Tools and Formats

These define how security information is presented, categorized by the audience's needs.

2.7.1.1 Reports

- **Core Concept:** Formal, static documents or exports that summarize security status, progress, or findings over a specific period (e.g., weekly, monthly, quarterly). They are typically used for auditing or compliance purposes.
- **Flow/Structure:** Reports often require manual review and context setting. They typically include executive summaries, detailed findings, remediation trends (**ART**, Domain 2.5), and compliance status.
- **Software Application Context:** A **Quarterly Compliance Report** detailing the percentage of critical applications that passed the latest penetration test and the status of outstanding **high-criticality vulnerabilities**.
- **Use Cases/Considerations:**
  - **Use When:** Communicating security status to auditors, compliance officers, or external regulators.
  - **Pros:** Provides a formal, time-stamped record of the security posture for **Accountability**.
  - **Cons:** Often lacks real-time relevance, as the data is historic; can be dense and require interpretation.

2.7.1.2 Dashboards

- **Core Concept:** Visual, often real-time, displays of key security metrics and indicators. Dashboards are designed for quick comprehension and continuous monitoring.
- **Flow/Structure:** They display **Key Performance Indicators (KPIs)** and **Key Risk Indicators (KRIs)** using charts and graphs. Dashboards are customizable based on the user's role (e.g., a "Developer Dashboard" vs. a "CISO Dashboard").
- **Software Application Context:** A dashboard might show the **Total Open Security Debt** (weighted by CVSS score), the **False Positive Rate** of SAST tools, and a real-time count of active security incidents.
- **Use Cases/Considerations:**

    - **Use When:** Providing instant visibility to technical teams (developers/SecOps) and executives for immediate decision-making.
    - **Pros:** Promotes continuous monitoring; highly effective for visualizing trends (e.g., Are we getting faster at fixing bugs?).
    - **Cons:** Can lead to **alert fatigue** or misprioritization if the wrong metrics are displayed.

2.7.2 Feedback Loops and Process Integration

These mechanisms ensure that security information is used to improve the underlying development processes.

2.7.2.1 Feedback Loops

- **Core Concept:** Mechanisms that ensure the output of security activities (e.g., testing, monitoring) is routed directly back to the relevant input phase (e.g., requirements, code implementation) to drive improvement and continuous refinement.
- **Flow/Structure:** This is a core component of the **DevSecOps** and **Agile** methodologies. It ensures that security knowledge gains from one phase are immediately applied to the next.
- **Software Application Context:** A vulnerability found during production monitoring (Domain 7) is immediately captured, analyzed for root cause, and leads to an update in the **Secure Coding Guidelines Documentation** (Domain 2.4) to prevent recurrence.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for continuous improvement in a mature security program (e.g., using SAMM/BSIMM).
    - **Pros:** Ensures lessons learned are permanently integrated into the process; helps prevent the re-introduction of old vulnerabilities (regression).
    - **Cons:** Requires strong coordination and a non-punitive culture where finding flaws is encouraged.

2.7.2.2 Integration with Development Tools

- **Core Concept:** Directly embedding security reports and findings into the tools developers use daily, minimizing context switching and speeding up remediation.
- **Flow/Structure:** Security findings (e.g., from SAST/DAST) are automatically converted into tickets in the development ticketing system (e.g., Jira), assigned a **Criticality Level** and an **Average Remediation Time** goal.

- **Software Application Context:** A security tool automatically comments on a **GitHub pull request**, flagging a vulnerability and preventing the code merge (**Break/Build Criteria**, Domain 2.3) until the fix is committed.
- **Use Cases/Considerations:**
    - **Use When:** Implementing **Shift Left** practices in a highly automated environment.
    - **Pros:** Accelerates the **ART** by eliminating manual steps; integrates security directly into the developer workflow.
    - **Cons:** Requires technical integration between disparate tools; a high rate of false positives can lead developers to ignore the feedback loop.

Domain 2.8 - Incorporate Integrated Risk Management Methods

2.8.1 Core Risk Terminology and Analysis

Definition

Risk is the possibility that a threat will exploit a vulnerability of an asset to cause harm to an organization. **Integrated Risk Management** (IRM) embeds this process into the entire software lifecycle to align security efforts with business objectives.

2.8.1.1 Key Risk Terms

- **Asset:** Something of value being protected (e.g., PII data, source code, web server).
- **Threat:** A potential cause of an undesirable incident (e.g., hacker, careless insider).
- **Vulnerability:** A weakness in a system or control that can be exploited by a threat (e.g., unpatched software, lack of input validation).
- **Exposure:** The state of being open to loss because of a threat agent.
- **Likelihood:** The probability that a given threat will successfully exploit a given vulnerability.
- **Impact:** The magnitude of harm that can be expected to result from a risk event (e.g., financial loss, reputational damage).

2.8.1.2 Inherent Risk vs. Residual Risk

- **Inherent Risk:** The level of risk that exists **before** any security controls or mitigation strategies have been implemented.
    - *Software Example:* A new API endpoint that accepts user-supplied JSON data and runs without any input validation or authentication.

- **Residual Risk:** The level of risk that remains **after** security controls have been implemented. Management must formally **accept** this remaining risk level.
    - **Formula:** Residual Risk = Inherent Risk - Impact of Controls
    - *Software Example:* The API endpoint now has strong input validation, but there is still a small, accepted risk that a zero-day vulnerability might be discovered in the underlying OS.

2.8.2 Risk Analysis and Measurement

2.8.2.1 Quantitative Risk Analysis

- **Core Concept:** Assigns objective monetary values to assets and potential losses, providing a specific, evidence-based justification for the cost of security controls (Cost-Benefit Analysis).
- **Flow/Structure:** Uses specific formulas to calculate the expected loss:
    - Single Loss Expectancy (SLE): The monetary loss expected if a specific threat exploits a vulnerability one time.
    - **Annualized Rate of Occurrence (ARO):** The expected number of times a specific threat is likely to occur in one year.
    - Annualized Loss Expectancy (ALE): The expected monetary loss per year from a specific risk. This is the maximum justifiable annual expenditure on a countermeasure.
- **Software Example using ALE:** If a proprietary source code repository has an ALE of **$320,000**, the organization can justify spending up to that amount per year on controls (e.g., DLP software, strong access controls).
- **Use Cases/Considerations:**
    - **Pros:** Results are specific and directly usable for **budget justification** and ROI calculations.
    - **Cons:** Difficult to assign accurate monetary values (AV, EF, ARO) to non-physical assets (e.g., data, reputation).

2.8.2.2 Qualitative Risk Analysis

- **Core Concept:** Uses subjective categories (Likelihood vs. Impact: Low, Medium, High) to rank risks based on expert opinion and scenario analysis.
- **Flow/Structure:** Risks are plotted on a **Risk Matrix**. This is essential for **Threat Modeling**.

- **Software Example:** A weak session cookie vulnerability is ranked **Critical** because of its **High Likelihood** (easy to exploit) and **High Impact** (full account takeover).
- **Use Cases/Considerations:**
  - **Pros:** Fast, cost-effective, and useful for quickly prioritizing remediation efforts.
  - **Cons:** Highly dependent on the expertise and bias of the risk assessor.

2.8.3 Control Implementation and Types

2.8.3.1 Risk Treatment Options

- **Mitigate (Reduce):** Implementing controls to lower the risk's likelihood or impact. *Example: Implementing input validation to reduce the likelihood of injection attacks.* (Most common option in SDLC)
- **Avoid:** Ceasing the activity that introduces the risk. *Example: Deciding not to store PII to avoid compliance risk.*
- **Transfer (Share):** Shifting the risk to another party. *Example: Buying cyber insurance.*
- **Accept:** Formally acknowledging the existence of the risk and deciding to take no action, usually because the cost of mitigation outweighs the ALE.

2.8.3.2 Control Categories (Function)

- **Preventive:** Stops a compromise from occurring. *Software Example: **Parameterizing database queries** to prevent SQL injection.*
- **Detective:** Identifies that a compromise has occurred. *Software Example: **Logging all failed authorization attempts** to a SIEM.*
- **Corrective:** Reduces the impact of a confirmed threat and restores the system. *Software Example: **Automated patching** or using an Incident Response Plan.*
- **Deterrent:** Discourages a threat from acting. *Software Example: Posting a clear security banner on the login page.*

2.8.4 Integrated Risk Management and Compliance

2.8.4.1 Technical Risk vs. Business Risk

- **Core Concept: Technical Risk** (flaws in code, scored by CVSS) must be contextualized by **Business Risk** (impact on finances, reputation, and compliance). **Business Risk** drives the final prioritization.
- **Software Example (Prioritization):**

    - A vulnerability in a legacy reporting service has a **Medium Technical Risk** (requires authentication).
    - However, the service contains EU customer PII, making the **Business Risk** of a GDPR violation **Critical**.
    - **Decision:** The vulnerability is elevated to **Critical Priority** because the potential fine far outweighs the technical difficulty of the exploit.

2.8.4.2 Regulations, Standards, and Guidelines

- **Core Concept:** Security planning must be derived from mandatory (Regulatory/Legal, Industry/Contractual) and voluntary (Best Practice) standards.
- **Examples:** Requires compliance with standards like **ISO**, **PCI**, **NIST**, **OWASP**, **SAFECode**, **SAMM**, and **BSIMM**. These standards provide the security controls necessary to mitigate risk.

2.8.4.3 Legal Compliance

- **Legal (Intellectual Property - IP):** Risk management must address IP protection by tracking all open-source and third-party components using a **Software Bill of Materials (SBOM)** to ensure licensing compliance.
- **Legal (Breach Notification):** The risk that a breach will require mandatory notification to affected individuals and/or regulatory bodies (e.g., GDPR, HIPAA). The **Incident Response Plan** (Domain 2.9) must contain the formal, timely notification procedures.

Domain 2.9 - Implement Secure Operation Practices

Definition: Establishing and executing formal procedures and governance mechanisms to ensure the continued security, integrity, and compliance of software systems once they are deployed into the operational environment.

2.9.1 Change Management Process

Definition

A formal, standardized procedure used to manage and control all modifications to the production environment, including software, hardware, and configuration settings. The goal is to ensure changes are necessary, authorized, tested, and documented, thereby preventing unintended negative impacts on security and availability.

- **Core Concept:** Maintaining the **Integrity** of the system by ensuring all modifications are tracked and approved (Accountability).

- **Flow/Structure:** A typical change process involves **Request -> Review/Security Impact Analysis -> Approval -> Implementation -> Verification -> Closure**. Security is a mandatory reviewer.
- **Software Application Context:** A request to update a third-party library to patch a critical vulnerability (Domain 7.9) must be submitted as a formal change request, reviewed by the security team for potential side effects, and verified by QA before deployment.
- **Use Cases/Considerations:**
    - **Use When:** Required for all production changes, especially those touching security-relevant files or configurations (e.g., firewall rules, database schema).
    - **Not Use When:** For emergency changes (e.g., active breach containment), which follow an accelerated, pre-approved process with mandatory post-change review.
    - **Pros:** Minimizes errors and security risks by enforcing **Segregation of Duties (SoD)**; provides a complete audit trail.
    - **Cons:** An overly bureaucratic or slow process can discourage necessary updates and lead to **security debt**.

2.9.2 Incident Response Plan (IRP)

Definition

A defined, documented set of procedures that guide the organization through the process of handling and recovering from a security breach or incident. The primary goals are containment, eradication, and returning to a secure, operational state as quickly as possible.

- **Core Concept:** The IRP is the **Corrective Control** that is executed when a detective control (e.g., logging) finds a failure.
- **Flow/Structure:** Common steps include: **Preparation -> Detection/Analysis -> Containment/Eradication -> Recovery -> Post-Incident Review (Lessons Learned)**. The IRP often triggers **Legal Breach Notification** procedures (Domain 2.8).
- **Software Application Context:** If a monitoring system detects unauthorized data exfiltration, the IRP dictates immediate actions: isolating the affected application instance (Containment), performing forensic analysis on memory and logs, and deploying the patched code (Eradication/Recovery).
- **Use Cases/Considerations:**
    - **Use When:** Any confirmed or suspected security violation occurs.
    - **Pros:** Minimizes damage (Impact) and recovery time (**Availability**); ensures a legally compliant response.

    - **Cons:** Requires regular testing (e.g., tabletop exercises) to ensure teams are familiar with the steps; an outdated plan can cause confusion during a real event.

2.9.3 Verification and Validation (V&V)

Definition

The process of ensuring the application meets its specified security requirements (**Validation**) and that the application is built correctly according to its design (**Verification**). In operations, V&V focuses on post-deployment checks.

- **Core Concept:** V&V ensures that security controls are not only implemented but are also effective in the live environment.
- **Flow/Structure:**
    - **Verification:** Checking that the system or process works as designed. *Example: Running a scan to confirm the OS patch deployed via the change management process was correctly applied.*
    - **Validation:** Checking that the final product meets the customer's *intent*. *Example: Performing a formal **Acceptance Test** to confirm the implemented authorization rules meet the business's **Need-to-Know** requirements.*
- **Software Application Context:** After a configuration change is deployed, a security engineer runs an automated test (Verification) to ensure the API still enforces the correct **Least Privilege** rules (Validation).
- **Use Cases/Considerations:**
    - **Use When:** Essential after any configuration or software change; mandatory before granting formal permission to operate.
    - **Pros:** Detects configuration drift and ensures that security flaws were not re-introduced during deployment (regression).
    - **Cons:** Can be limited if the **Security Requirements** (Domain 3) were poorly defined initially.

2.9.4 Assessment and Authorization (A&A) Process

Definition

A formal, regulatory-driven process where a security system is rigorously evaluated (**Assessment**) against a set of standards (e.g., NIST SP 800-53), and a senior management official formally assumes the remaining risk and grants an **Authority to Operate (ATO)** (**Authorization**).

- **Core Concept:** This is the final **Authorization** gate (Domain 2.8) that provides **Accountability** for the system's security posture.

- **Flow/Structure:** A&A is the culmination of the entire NIST Risk Management Framework (RMF). It requires extensive documentation (Domain 2.4), a formal **Risk Assessment** (Domain 2.8), and a sign-off by a designated authorizing official (AO).
- **Software Application Context:** Before a new financial system goes live, the CISO (or AO) reviews the penetration test results, the residual risk documented in the risk register, and the Incident Response Plan, then issues the ATO.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for government systems (federal, defence) and systems handling highly sensitive data (e.g., federal health records).
    - **Pros:** Ensures executive-level visibility and acceptance of the system's **Residual Risk**; establishes high **Accountability**.
    - **Cons:** Can be a time-consuming and documentation-heavy process, often seen as a compliance hurdle rather than a development activity.

Domain 3 – Secure Software Requirements – 13%

Domain 3.1 - Define Software Security Requirements

Definition: The systematic process of gathering, analyzing, and documenting all necessary capabilities and constraints (both functional and non-functional) that an application must satisfy to achieve its business objectives and required security posture.

3.1.1 Functional Security Requirements (FSRs) - The "What" of Security

3.1.1.1 Definition and Core Concept

- **Definition:** FSRs define the explicit, verifiable security **features** that the software must possess and that users or administrators directly interact with or rely on for protection. [1]
- **Core Concept:** How the system implements a security *feature* as part of its designed business logic. These are typically derived from **business requirements**, **use cases**, or **stories**. [2] FSRs are verifiable with a simple pass/fail outcome.

Attribute	Description
Origin	Business requirements, high-level policies. [3]
Verifiability	High (Pass/Fail).
Example	"The system **shall** log a security event upon three failed login attempts." [4]

3.1.1.2 Conversion to Use Cases

- **Core Concept:** A **use case** describes a sequence of interactions between an **actor** and the system to achieve a specific goal. Functional security requirements are implemented as steps, conditions, or rules *within* the overall business use case.
- **Flow/Example:** FSRs often define the **failure conditions** that prevent the goal from being achieved. For example, a "Transfer Funds" use case incorporates the FSR rule: The system validates the transfer amount against the user's privilege level; if the amount exceeds the limit, the system follows an **extension point** and displays an "Authorization Denied" error.

3.1.1.3 Conversion to User Stories (Agile/Scrum)

- **Core Concept:** A **user story** is an informal statement of a feature written from the end-user's perspective. FSRs are often converted into **security stories** or added as non-negotiable **acceptance criteria** to existing functional stories in the Agile backlog.
- **Structure (Security Story):** Security requirements must be consumable by the development team.
    - *Standard Story Example:* "As a customer, I want to log in quickly so I can access my account."
    - *FSR -> Acceptance Criteria:* The system must verify the file extension and header signature of uploaded images before saving them (implementing the FSR: "The system shall restrict file uploads to non-executable file types").
- **Use Cases/Considerations:**
    - **Use When:** Working within an Agile framework.
    - **Pros:** Forces the team to prioritize and size security work alongside business features; ensures security is part of the **Definition of Done**.
    - **Cons:** FSRs can be overlooked if they are not explicitly written as acceptance criteria in the user story.

3.1.2 Non-Functional Security Requirements (NFRs) - The "How Secure" of Security

Definition

Non-functional requirements define the constraints, qualities, and characteristics of the system, defining the expected level of resilience, performance, and robustness for security controls. [5]NFRs are derived from **Risk Assessments** (Domain 2.8) and compliance standards (Domain 2.2). [6]

3.1.2.1 Security Requirements

- **Core Concept:** NFRs directly related to preventing compromise, defining acceptable cryptographic strength, error handling, and session management. [7]
- **Software Application Context:**
    - "The system **must** protect data in transit using **TLS 1.2 or higher**."
    - "The system **must** securely hash passwords using PBKDF2 with at least 100,000 iterations."
    - "The system **must** fail securely (Fail Secure) if the database connection is lost."
- **Use Cases/Considerations:**

    - **Use When:** Mandatory for ensuring the technical mechanisms underpinning security features are robust.
    - **Pros:** Defines the **minimum security baseline** needed to mitigate high-likelihood risks.

3.1.2.2 Operational Requirements

- **Core Concept:** NFRs that define the system's ability to run, be managed, and be monitored securely in the production environment. [8]
- **Software Application Context:**
    - "The system **must** generate detailed security audit logs for all authentication and authorization failures (Accountability)."
    - "The system **must** support deployment using automated **Infrastructure as Code (IaC)** tools."
- **Use Cases/Considerations:**
    - **Use When:** Planning for secure operations, logging, monitoring, and patch management (Domain 7).
    - **Pros:** Ensures the system is manageable and observable (telemetry) during security incidents.

3.1.2.3 Continuity Requirements (Resiliency/Availability)

- **Core Concept:** NFRs that ensure the system can maintain essential functions during disruptions and recover within specified timeframes. [9]
- **Software Application Context:**
    - "The system **must** achieve a 99.999% uptime (**Availability**)."
    - "The system **must** meet a **Recovery Time Objective (RTO)** of less than 4 hours following a full site failure (Disaster Recovery)."
    - "The system **must** use geo-redundant storage for all data (Resiliency)."
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for mission-critical applications where downtime incurs significant financial or safety risk.
    - **Pros:** Essential for BCP/DR planning; provides a measurable goal for high availability.

3.1.2.4 Deployment Requirements

- **Core Concept:** NFRs that specify the constraints and environment parameters necessary to securely build, package, and deploy the software. [10]
- **Software Application Context:**
    - "The deployment artifact **must** be cryptographically signed by the CI/CD pipeline before being accepted by the production cluster."
    - "The application **must** be deployable via automated blue/green deployment strategy."
- **Use Cases/Considerations:**
    - **Use When:** Designing the CI/CD pipeline and enforcing supply chain security (Domain 8).
    - **Pros:** Ensures deployment is repeatable, verifiable (**integrity**), and minimizes manual errors.

Domain 3.2 - Identify Compliance Requirements

Definition: The process of identifying, documenting, and incorporating all external and internal security mandates, laws, standards, and policies that govern the software, its data, and its development process. These requirements directly translate into non-functional security constraints (Domain 3.1).

3.2.1 External Regulatory and Legal Requirements

These are mandatory rules imposed by governments or legal authorities, carrying the risk of fines, penalties, or legal action if violated (Legal, Domain 2.8).

3.2.1.1 Regulatory Authority

- **Core Concept:** Laws and regulations issued by governmental bodies that dictate specific security and privacy practices for data and systems.
- **Flow/Structure:** These mandates define the types of data that must be protected (e.g., PII) and set deadlines for breach notification.
- **Software Application Context:** Compliance with the **General Data Protection Regulation (GDPR)** requires the application to implement explicit **User Rights** (e.g., right to be forgotten, right to data portability) and **Cross-Border Transfer Controls** (Domain 3.4).
- **Use Cases/Considerations:**
    - **Use When:** The application processes data belonging to citizens of a regulated jurisdiction (e.g., EU, California).
    - **Pros:** Ensures legal operations and protects the organization from severe fines.

    - **Cons:** Requirements are jurisdiction-specific and constantly changing, necessitating continuous legal review.

3.2.1.2 Legal

- **Core Concept:** Broad legal obligations that affect the software, including liability, intellectual property (IP), and mandated disclosures.
- **Flow/Structure:** Requirements often define legal clauses that must be enforced by the software (e.g., End-User License Agreements (EULA)).
- **Software Application Context:** The software must include mechanisms to enforce **IP** protection for proprietary algorithms and comply with third-party licensing terms for open-source components (**Software Supply Chain**, Domain 8).
- **Use Cases/Considerations:**
    - **Use When:** Using third-party components or selling software commercially.
    - **Pros:** Protects organizational assets and avoids complex legal disputes.

3.2.2 External Industry-Specific Requirements

These are standards imposed by industry organizations or consortia, often required for business operations.

3.2.2.1 Financial (e.g., Payment Card Industry - PCI)

- **Core Concept:** Standards necessary for processing, storing, or transmitting financial data. **PCI DSS** is the most common mandate in this category.
- **Flow/Structure:** PCI DSS requires specific security controls, such as strict cryptographic protocols for transmission, regular penetration testing, and secure coding practices (e.g., Requirement 6).
- **Software Application Context:** An e-commerce payment module must isolate its environment (**segmentation**) and use only approved, current encryption for all cardholder data (CHD).
- **Use Cases/Considerations:**
    - **Use When:** The application handles credit card data.
    - **Pros:** Necessary for maintaining the ability to process payments; reduces liability for fraud.

3.2.2.2 Healthcare (e.g., HIPAA)

- **Core Concept:** Regulations governing the protection of sensitive patient health information (PHI) and electronic health records (EHRs).

- **Software Application Context:** Access controls must be implemented to ensure only authorized personnel can view PHI (Need-to-Know), and all access must be logged for auditing.
- **Use Cases/Considerations:**
    - **Use When:** Developing software for healthcare providers, insurance companies, or any entity handling PHI.

3.2.2.3 Defense and Commercial

- **Core Concept:** Security requirements specific to government contracting or critical infrastructure (e.g., NERC CIP for electric grid). Defense contracts often require compliance with standards like **NIST SP 800-171** to protect controlled unclassified information (CUI).
- **Software Application Context:** A defense contractor's software must implement specific **Assessment and Authorization (A&A)** processes and use FIPS-validated cryptography.
- **Use Cases/Considerations:**
    - **Use When:** The application is used in critical infrastructure or government supply chains.

3.2.3 Internal Company-Wide Requirements

These are internal policies and standards created to implement and enforce external mandates consistently across the organization.

3.2.3.1 Development Tools and Standards

- **Core Concept:** Internal rules defining the permissible tools, methodologies, and security posture for software creation.
- **Flow/Structure:** These are often documented in the **Coding Standards Documentation** (Domain 2.4).
- **Software Application Context:** A policy might mandate that all new code must be written in a specific language (e.g., Python 3.x), must use a centralized credential management vault for secrets, and must achieve a minimum of 80% code coverage in unit testing.
- **Use Cases/Considerations:**
    - **Use When:** Standardizing development practices across multiple teams.
    - **Pros:** Enforces consistency, simplifies auditing, and reduces the complexity introduced by tool sprawl (**Economy of Mechanism**).

3.2.3.2 Frameworks and Protocols

- **Core Concept:** Internal mandates on the use of specific security frameworks and communication protocols.
- **Flow/Structure:** Ensures inter-application communication and authentication are secure and standardized.
- **Software Application Context:** A policy may require all new microservices to use **OAuth 2.0** (protocol) for authorization and **TLS 1.3** for all internal communication. The application development must follow the steps defined by the **OWASP SAMM** framework.
- **Use Cases/Considerations:**
  - **Use When:** Integrating new applications into a large enterprise ecosystem.
  - **Pros:** Reduces attack surface by eliminating older, weaker protocols; promotes **Component Reuse** for secure modules.

Domain 3.3 - Identify Data Classification Requirements

Definition: The systematic process of organizing and labelling data based on its **sensitivity** and **impact** (value) to the organization. Classification dictates the necessary security controls, handling procedures, and storage requirements throughout the data's entire lifecycle.

3.3.1 Data Governance and Ownership

Definition

Establishing clear roles and responsibilities for managing data assets, ensuring accountability for their protection and use.

3.3.1.1 Data Owner

- **Core Concept:** The individual or role (usually a manager or executive) who has the ultimate **responsibility and accountability** for the data's protection, classification, and authorization for use. They decide the data's sensitivity and acceptable risk level.
- **Software Application Context:** The **Vice President of Human Resources** is the Data Owner for all employee salary information, mandating its classification as "Restricted."
- **Use Cases/Considerations:**
  - **Pros:** Enforces high-level **Accountability** (Domain 1.1.6); ensures data security aligns with business risk tolerance (Domain 2.8).

3.3.1.2 Data Custodian

- **Core Concept:** The individual or role (usually IT or Development staff) responsible for the **technical implementation and maintenance** of the security controls (encryption, backup, access controls) mandated by the Data Owner. They manage the data on a daily basis.
- **Software Application Context:** A **Database Administrator (DBA)** or **Application Developer** who is responsible for configuring the database encryption keys and access permissions for the data.
- **Use Cases/Considerations:**
    - **Pros:** Separates the **duty** of defining security policy (Owner) from implementing it (Custodian) (**Segregation of Duties**, Domain 1.2.2).

3.3.1.3 Data Dictionary

- **Core Concept:** Documentation that provides standardized definitions, storage formats, and the assigned **classification level** for every data element used within the application.
- **Software Application Context:** The dictionary entry for the 'Customer Social Security Number' field explicitly lists its format and its classification (Restricted/PII).
- **Use Cases/Considerations:**
    - **Pros:** Ensures consistency across all applications and databases; crucial for automated tools like **Data Loss Prevention (DLP)** systems.

3.3.2 Data Labeling and Types

Definition

Labeling assigns a formal category (classification) to data, which automatically determines the required security controls.

3.3.2.1 Data Labeling (Sensitivity and Impact)

- **Core Concept:** Assigning one of a predetermined set of classification labels (e.g., Public, Internal, Confidential, Restricted) based on the data's **sensitivity** and the **impact** to the organization if the data is disclosed or corrupted.
- **Flow/Structure:** The higher the sensitivity, the stricter the controls (e.g., **Restricted/Secret** requires mandatory segmentation and FIPS-validated encryption).
- **Software Application Context:** A **Data Masking** routine is applied to all fields labeled "Confidential" when accessed by non-administrative users.

3.3.2.2 Data Types (Structured, Unstructured)

- **Core Concept:** Identifying the format of the data, as the format dictates the appropriate security control methods.
- **Structured Data:** Highly organized data that resides in a fixed field (e.g., SQL database). *Security Control:* **Field-level encryption**, **Row-level access control**.
- **Unstructured Data:** Data that does not have a predefined format (e.g., email bodies, documents). *Security Control:* **Whole-file encryption**, **DLP pattern matching**.
- **Use Cases/Considerations:**
    - **Pros:** Ensures the correct technical controls are applied; for example, you cannot easily apply row-level access control to an unstructured PDF file.

3.3.3 Data Lifecycle and Handling

Definition

Security controls must be applied at every phase of the data's existence, from creation to ultimate destruction.

3.3.3.1 Data Lifecycle (Generation, Storage, Retention, Disposal)

- **Core Concept:** The sequence of defined stages a piece of data goes through from creation to disposal. Security must be managed in all three data states: **in use (processing)**, **in transit (moving)**, or **at rest (storage)**.
- **Flow/Structure: Stages and Security Focus:**
    - **Generation/Collection:** (In Use) **Focus:** Input validation, integrity of data source, consent capture.
    - **Storage:** (At Rest) **Focus:** Preventing unauthorized access over time. **Control:** Encryption at rest, database access controls (Domain 4).
    - **Processing/Use:** (In Use) **Focus:** Protecting data while it is loaded into memory. **Control:** Memory protection (e.g., ASLR), least privilege execution.
    - **Transfer:** (In Transit) **Focus:** Protecting data integrity and confidentiality while moving between systems. **Control:** Encrypted protocols (**TLS/VPN**), integrity checks (**hashing**).
    - **Retention/Archival:** (At Rest) **Focus:** Secure long-term storage according to policy (Domain 2.6).
    - **Disposal/Destruction:** (Various) **Focus:** Irreversible elimination of data. **Control:** Cryptographic erasure, physical destruction (Domain 2.6).

- **Software Application Context:** During the **Storage** phase, the application uses AES-256 for data at rest. During the high-risk **Processing/Use** phase, memory is protected using OS controls.
- **Use Cases/Considerations:**
    - **Pros:** Ensures security is continuous and layered (**Defense in Depth**); allows controls to be tailored to the data's most vulnerable state.
    - **Cons:** The "Data in Use" phase is often the most challenging to secure because the data must be in a decrypted state for processing.

3.3.3.2 Data Handling (PII, Publicly Available Information)

- **Core Concept:** Specific legal and security procedures that must be followed when processing certain types of data, particularly those with regulatory requirements.
- **Personally Identifiable Information (PII):** Data that can be used to identify, contact, or locate a single person (e.g., name, address, SSN). Requires strong **Confidentiality** and **Privacy** controls (Domain 3.4).
    - **Software Application Context:** PII requires **Anonymization** (Domain 3.4) before being used in a non-production test environment.
- **Publicly Available Information:** Data that can be freely accessed by the general public. Requires strong **Availability** and **Integrity**, but minimal confidentiality controls.
- **Use Cases/Considerations:**
    - **Pros:** Ensures compliance with privacy laws (GDPR, CCPA) by providing appropriate technical controls for PII.
    - **Cons:** Over-classifying non-PII data as PII can increase compliance costs unnecessarily.

Domain 3.4 - Identify Privacy Requirements

Definition: The process of identifying, documenting, and implementing specific technical and procedural requirements necessary to comply with legal mandates for protecting Personally Identifiable Information (PII) and safeguarding the privacy rights of the data subject. This extends beyond security (CIA) into legal use and disclosure.

3.4.1 Data Collection and Minimization

Definition

Defining the scope of data collected and ensuring the principle of **Data Minimization** is followed—only collecting data that is strictly necessary for the specified purpose.

3.4.1.1 Data Collection Scope

- **Core Concept:** Explicitly defining *what* data elements are collected, *why* they are collected (the legal basis), and *how* long they will be kept. Collection must be lawful, fair, and transparent.
- **Flow/Structure:** The scope is typically reviewed by legal/compliance to ensure alignment with the **Privacy Policy** and **Data Classification Policy** (Domain 3.3).
- **Software Application Context:** The application should only collect a user's geographical location if that information is strictly needed to fulfill a location-based service requested by the user, and not for unsolicited marketing purposes.
- **Use Cases/Considerations:**
    - **Use When:** Any data element is gathered, especially PII. **Mandatory** for demonstrating compliance with GDPR and CCPA.
    - **Pros:** Reduces the attack surface and minimizes organizational liability by storing less sensitive data.
    - **Cons:** Overly broad or undocumented collection scopes violate privacy laws.

3.4.2 Data Anonymization and De-identification

Definition

The process of transforming PII to prevent the data subject's re-identification, a crucial step for using sensitive data in non-production environments (e.g., development, testing).

3.4.2.1 Pseudo-Anonymous

- **Core Concept:** Replacing direct identifiers (like names or SSNs) with **reversible pseudonyms** (tokens or unique keys). Re-identification is possible by linking the pseudonym back to the original identifier using a separate, secure key management system.
- **Flow/Structure:** Data is **tokenized** (Domain 5). The link key is protected by strong encryption and strict **Least Privilege** access controls, typically isolating it from the pseudo-anonymous data set.
- **Software Application Context:** A customer's full address is replaced with a random token in a test database. The key to reverse the tokenization is stored in a segregated secrets vault, accessible only to compliance auditors.
- **Use Cases/Considerations:**
    - **Use When:** Data needs to be used for testing/analytics but must still maintain integrity for specific customer lookups if necessary.

    - **Cons:** Still considered PII under many regulations (like GDPR) because re-identification is technically possible, requiring ongoing security controls.

3.4.2.2 Fully Anonymous

- **Core Concept:** Irreversibly altering PII such that the data subject **cannot be identified** by any means. This often involves techniques like **data masking**, **data shuffling**, or **aggregation**.
- **Flow/Structure:** Data is aggregated (summed or averaged) or direct identifiers are permanently scrubbed. The integrity of the business logic must be maintained while confidentiality is maximized.
- **Software Application Context:** Calculating the average purchase size for customers in a specific zip code over a month, thus eliminating individual purchase records and names.
- **Use Cases/Considerations:**
    - **Use When:** Creating training or analytical data sets that must be used outside of the secure production environment.
    - **Pros:** Once truly anonymous, the data is no longer subject to many privacy regulations.
    - **Cons:** The transformation is irreversible, and technical methods must be used to mitigate the risk of **re-identification via correlation** (linking multiple anonymous data points).

3.4.3 User Rights and Preferences (Legal)

Definition

Implementing technical capabilities that allow data subjects to exercise their legal rights over their data, as mandated by laws like GDPR (Data Subject Rights).

3.4.3.1 Data Disposal / Right to be Forgotten

- **Core Concept:** The legal obligation to permanently and securely **delete** a data subject's PII upon request, as long as no overriding legal reason for retention exists.
- **Flow/Structure:** The application must have a documented process (Data Disposal Policy, Domain 2.6) and a technical mechanism (API) to trigger the irreversible deletion of all PII associated with a user ID from production, archive, and backup systems.
- **Software Application Context:** A user submits a request to be forgotten via the application's account settings, which triggers a workflow to locate and cryptographically erase all associated PII, while retaining only necessary, non-identifying transaction data (e.g., for financial auditing).

3.4.3.2 Marketing Preferences and Sharing/Third Parties

- **Core Concept:** The requirement for the application to capture, store, and enforce the user's explicit consent and preferences regarding how their data is used for non-essential purposes (e.g., marketing, selling data to third parties).
- **Software Application Context:** The application must present a clear, opt-in consent form and reliably restrict the export of a user's PII if they have opted out of third-party sharing.
- **Pros:** Ensures transparency and compliance with consent mandates.

3.4.3.3 Terms of Service (ToS)

- **Core Concept:** The legally binding agreement between the service provider and the user. The privacy requirements are often derived from and must be consistent with the promises made in the ToS.
- **Software Application Context:** Any security or data handling feature implemented must not contradict the promises made in the application's ToS regarding data security and privacy.

3.4.4 Data Geography and Retention

Definition

Requirements governing where data may be physically stored and processed, and for how long.

3.4.4.1 Data Retention (How Long, Where, What)

- **Core Concept:** The length of time data must be legally or functionally kept (Domain 2.6). This is a critical factor in risk management, as the longest-retained data represents the longest-lasting risk.
- **Flow/Structure:** Privacy mandates typically define the *maximum* retention period (e.g., PII should not be kept indefinitely), while other laws (e.g., tax) define the *minimum* retention period.
- **Software Application Context:** A developer designs the application to automatically move PII to a secure archive and delete the primary record after two years of account inactivity, adhering to the principle of data minimization.

3.4.4.2 Cross-Border Requirements (Data Residency, Jurisdiction)

- **Core Concept:** Legal restrictions on the physical geographic location where data can be stored, processed, or accessed, especially PII crossing national or regional borders.

- **Data Residency:** The data must reside in a specific country or region (e.g., Canada requires certain health data to remain on Canadian soil).
- **Jurisdiction:** Which country's laws apply to the data, often determined by the data subject's location or the processing location.
- **Software Application Context:** The application architecture must include **geo-segmentation** controls, ensuring PII collected from European users is only stored and processed within EU-based data centers (Multi-National Data Processing Boundaries).
- **Use Cases/Considerations:**
  - **Use When:** Mandatory for multinational applications.
  - **Cons:** Requires complex, separate infrastructure deployments (e.g., distinct cloud regions) for different user groups, adding complexity and cost.

Domain 3.5 - Define Data Access Provisioning

Definition: The systematic process of creating, modifying, managing, and terminating access rights and accounts (identities) for both human users and automated services. Proper provisioning ensures that access is granted only when required and revoked immediately when no longer needed.

3.5.1 User Provisioning (Humans)

Definition

The process of managing the lifecycle of access rights for **human users** based on their roles and employment status (**Role-Based Access Control - RBAC**).

- **Core Concept:** The lifecycle of a human user's access: **Creation -> Modification/Review -> Revocation/Deprovisioning**. Must enforce the principle of **Least Privilege**[1111].
- **Flow/Structure:** Often tied to HR systems (Identity and Access Management - **IAM**) to automate access creation on hiring (provisioning) and removal on termination (**deprovisioning**). Access grants are based on the user's role (e.g., "Developer," "Auditor").
- **Software Application Context:** When a new developer is hired, the system automatically provisions them with a "Developer" role, granting them **read-only** access to non-production databases and **read/write** access to their assigned source code repository.
- **Use Cases/Considerations:**
  - **Use When:** Managing any human user access, especially for systems handling sensitive data (PII, financial).

  - **Pros:** Enforces consistency and drastically reduces the risk of orphaned accounts (accounts belonging to terminated employees), which are a major attack vector.
  - **Cons:** Over-relying on generic roles (e.g., granting "Super-User" access broadly) violates the principle of **Least Privilege**.

3.5.2 Service Accounts (Non-Humans)

Definition

Accounts used by automated processes, applications, or microservices (non-human entities) to interact with other resources, such as databases, secrets managers, or external APIs.

- **Core Concept:** Service accounts must adhere even more strictly to the **Least Privilege** principle than human accounts, as they are typically non-interactive and often lack immediate human monitoring.
- **Flow/Structure:** Service accounts should be strictly mapped to a single function or component (e.g., one microservice gets one dedicated service account). They should use non-password credentials (e.g., **API keys, short-lived tokens, or certificates**) and be automatically rotated.
- **Software Application Context:** A specific microservice used for logging receives a unique service account with **INSERT-ONLY** privileges for the log database table and no permissions for any other tables (**Least Privilege**). The key for this account is stored in a secrets vault.
- **Use Cases/Considerations:**
  - **Use When:** Any time one application or service needs to interact with another system.
  - **Pros: Limits the blast radius** (Domain 1.2.8) if a component is compromised, as the attacker only gains the service's very limited permissions.
  - **Cons:** Manual creation and tracking of numerous service accounts can lead to sprawl, necessitating automated **secrets management**.

3.5.3 Reapproval Process (Recertification/Review)

Definition

A formal, periodic procedure (often mandated by compliance) that requires the **Data Owner** (Domain 3.3) to review and explicitly re-authorize access rights for all users (human and service accounts).

- **Core Concept:** Ensures that access rights, even if initially correct, are **still necessary** and have not drifted over time (e.g., a user changed jobs but kept old permissions). This is a critical administrative control for managing **residual risk**.

- **Flow/Structure:** The process requires the Data Owner to review a list of all current users/service accounts with access to their data and affirm one of three options: **Approve, Modify, or Revoke**. This is often automated annually or semiannually.
- **Software Application Context:** The Data Owner for the customer PII database receives an alert every six months listing all 50 people and 10 service accounts with access. They must formally check a box confirming that the access is still required.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for systems dealing with regulatory compliance (e.g., Sarbanes-Oxley, PCI).
    - **Pros:** Detects and eliminates unnecessary, excessive, or stale permissions (**Permission Creep**), enforcing the continued principle of **Least Privilege**.
    - **Cons:** Can create friction if owners are slow to respond, potentially leading to unnecessary revocation of required access.

Domain 3.6 - Develop Misuse and Abuse Cases

Definition: The structured process of identifying ways an application can be deliberately attacked (**Abuse Cases**) or improperly used (**Misuse Cases**) by hostile or unauthorized actors. This is an attacker-centric approach to requirements gathering that ensures developers proactively build controls to counter known threats (a key step in **Threat Modeling**).

3.6.1 Misuse Cases and Abuse Cases

Definition

Formal descriptions of a system's interaction with a malicious actor (or "mis-actor"). They are the negative counterparts to standard **Use Cases** (Domain 3.1).

3.6.1.1 Misuse Case

- **Core Concept:** Describes a situation where an authenticated, legitimate user attempts to use the system in a way it was not intended, typically to bypass security or business rules. The actor is usually an **Insider Threat**.
- **Software Application Context:** A customer uses a web proxy to intercept and modify the HTTP request payload after the application has calculated a 10% fee, changing the fee amount to zero before the data hits the server.
- **Use Cases/Considerations:**
    - **Use When:** Identifying flaws in business logic and authorization enforcement.
    - **Pros:** Highly effective for detecting vulnerabilities related to trust boundaries, business rule enforcement, and **input validation** (Domain 5).

3.6.1.2 Abuse Case

- **Core Concept:** Describes a situation where an external, unauthenticated, or anonymous actor attempts to deliberately attack the system using technical vulnerabilities. This often leads to compromise of **Confidentiality** or **Integrity**.
- **Software Application Context:** An external attacker attempts to exploit a search form by entering a malicious string (' OR 1=1 --) designed to bypass authentication and dump the entire user table.
- **Use Cases/Considerations:**
  - **Use When:** Mandatory for all public-facing interfaces and high-risk API endpoints.
  - **Pros:** Directly addresses external threats and maps known vulnerabilities to specific design elements.

3.6.2 Mitigating Control Identification and Process

Definition

The process of identifying specific security controls and documenting them as formal requirements to block the identified misuse and abuse scenarios. This is where risk analysis translates directly into development tasks.

3.6.2.1 The Three-Step Process (CSSLP Focus)

The CSSLP emphasizes the structured, repeatable process of generating controls:

1. **Identify the Misuse/Abuse Case:** Define the malicious scenario (e.g., "Anonymous User Injects Malicious Code").
2. **Identify the Vulnerability:** Determine the specific software flaw that makes the attack possible (e.g., lack of input validation).
3. **Define the Mitigating Control (Requirement):** Create a specific, technical or procedural control (Functional or Non-Functional Requirement) to eliminate or reduce the risk.

3.6.2.2 Utilizing External Lists (Resources)

Security engineers use external catalogs of threats to **inform** the process, rather than relying on a static master list:

- **OWASP Top 10:** Directly informs the most relevant **Abuse Cases** (e.g., A1: Broken Access Control).
- **Common Weakness Enumeration (CWE):** A dictionary of **software weaknesses** (e.g., CWE-89: Improper Neutralization of Special Elements) used to quickly **identify the vulnerability** that enables the abuse.

- **OWASP ASVS (Application Security Verification Standard):** The specific security requirements defined in ASVS are often chosen as the mitigating controls themselves (e.g., an ASVS requirement for strong session handling mitigates a session hijacking abuse case).

3.6.2.3 Documenting the Mitigating Control

- **Core Concept:** For every identified risk, at least one **Preventive Control** must be designed and documented as a security requirement.
- **Software Application Context (Example):**
    - **Abuse Case:** SQL Injection.
    - **Mitigating Control 1 (Non-Functional Requirement - Code):** "The application **shall** use **parameterized database queries** for all user-supplied input."
    - **Mitigating Control 2 (Non-Functional Requirement - Authorization):** "The application database service account **shall** only have **Least Privilege** access (no DROP/ALTER table rights)."
- **Use Cases/Considerations:**
    - **Use When:** This technique is a key element of the **Shift Left** process, ensuring security is **baked in** at the design phase.
    - **Pros:** Directly links security requirements to specific, quantifiable risks, making it easier to justify development costs.
    - **Cons:** Mitigating controls must be layered (**Defense in Depth**); relying on a single control (e.g., only a WAF) leaves the application vulnerable if that control is bypassed.

Domain 3.7 - Develop Security Requirement Traceability Matrix

Definition: A table that maps and links high-level requirements (e.g., compliance mandates) to low-level artifacts (e.g., design components, test cases). The **Security Requirement Traceability Matrix (SRTM)** is a crucial tool for governance, ensuring that every security requirement is addressed and tested.

3.7.1 Security Requirement Traceability Matrix (SRTM)

Definition

A document that ensures the complete fulfillment of all security requirements throughout the SDLC. It provides a formal, verifiable path from origin (why the requirement exists) to outcome (how the requirement was tested and met).

- **Core Concept:** The SRTM provides an auditable path showing that no security requirements were forgotten, implemented incorrectly, or left untested (**Complete Mediation** on the process level).
- Flow/Structure: The matrix includes columns that map artifacts across multiple phases of the SDLC:
- **Software Application Context:** The matrix shows that **GDPR's Right to be Forgotten** (Source Mandate) led to the FSR: "The system must securely delete all PII within 24 hours of request" (Security Requirement). This FSR is linked to the "Data Purge API" (Design Component) and verified by a "Deletion Acceptance Test" (Test Case).
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for applications subject to formal compliance audits (e.g., NIST A&A process, PCI).
    - **Pros: Guarantees coverage** by proving every requirement is tested; simplifies gap analysis during audits.
    - **Cons:** Time-consuming to maintain manually, especially in fast-paced Agile environments; requires continuous updates whenever requirements or code changes (Change Management).

3.7.2 Traceability Links and Types

Definition

The connections established within the matrix that define the relationships between artifacts. Security requires both **forward** and **backward** traceability.

3.7.2.1 Forward Traceability

- **Core Concept:** Tracing artifacts from the beginning of the SDLC forward to the end. It ensures that every **requirement** leads to working code and a successful test.

3.7.2.2 Backward Traceability (Reverse)

- **Core Concept:** Tracing artifacts from the end of the SDLC backward to the origin. It ensures that every artifact (code or test) can be justified by a valid requirement.

3.7.2.3 Two-Way Traceability

- **Core Concept:** Achieving both forward and backward traceability, establishing a clear link between every security requirement and its final test result.

Domain 3.8 - Define Third-Party Vendor Security Requirements

Definition: The process of establishing, communicating, and enforcing security expectations and controls for any external software, components, or services acquired from third-party vendors. This extends security beyond the organization's immediate boundaries to manage **Software Supply Chain Risk**.

3.8.1 Software Supply Chain and Acquisition

Definition

Managing the security of all components and services, from their origin through delivery, that contribute to the final software product.

3.8.1.1 Software Supply Chain

- **Core Concept:** The entire ecosystem involved in creating, distributing, and managing software, including developers, build environments, third-party libraries, open-source components, and deployment platforms. Security breaches (e.g., SolarWinds) often occur here.
- **Flow/Structure:** Requires a comprehensive **Software Bill of Materials (SBOM)** (Domain 8.2) to track all dependencies and their known vulnerabilities.
- **Software Application Context:** The organization must secure its CI/CD pipeline, ensuring that all third-party libraries pulled into the build are scanned for vulnerabilities (e.g., using **Software Composition Analysis - SCA** tools, Domain 7.9) and originate from trusted registries.
- **Use Cases/Considerations:**
  - **Use When:** Any software uses external components or is built using external tools/services.
  - **Pros:** Essential for reducing **third-party risk**; proactively prevents the introduction of known vulnerabilities into the application.
  - **Cons:** Can be complex to manage due to the deep dependency trees of modern software.

3.8.1.2 Software Component Acquisition

- **Core Concept:** Defining security requirements based on the type of software component being acquired, as each type carries different risks and requires specific due diligence.
- **Commercial Off-the-Shelf (COTS):** Pre-built software purchased from a vendor (e.g., a commercial database, an enterprise application).
  - *Requirements:* Review vendor's security documentation, **penetration test reports**, certifications (e.g., SOC 2, ISO 27001).

- **Open Source:** Publicly available software components (e.g., libraries, frameworks).
    - *Requirements:* Perform **SCA** to identify known vulnerabilities (CVEs), review license compatibility, check community support and maintenance.
- **Cloud Services (SaaS, PaaS, IaaS):** Software-as-a-Service, Platform-as-a-Service, Infrastructure-as-a-Service.
    - *Requirements:* Understand the **Shared Responsibility Model** (who secures what), review vendor's security controls, audit reports, and certifications.
- **Partners:** Software developed or hosted by a business partner.
    - *Requirements:* Establish **inter-organizational security agreements** (e.g., BAAs, MOUs), conduct joint security reviews.
- **Use Cases/Considerations:**
    - **Pros:** Tailors security due diligence to the specific risk profile of each component.
    - **Cons:** Due diligence can be challenging for open-source components with limited documentation.

3.8.2 Vendor Attestation and Assurance

Definition

Methods used to gain confidence that a third-party vendor's security practices meet the required standards.

3.8.2.1 Attestation (Audit Reports, Certifications)

- **Core Concept:** Formal evidence provided by a third-party vendor (often through an independent auditor) to confirm that their security controls and processes meet specific standards.
- **Flow/Structure:** These are typically standardized reports.
    - **SOC 2 Type 2 Report:** Details the design and operating effectiveness of a vendor's security controls over a period of time.
    - **ISO 27001 Certification:** Confirms an organization has implemented an Information Security Management System (ISMS).
- **Software Application Context:** Before integrating a cloud-based CRM, the organization requests its **SOC 2 Type 2 report** to verify its security posture and ensure it meets internal requirements for data protection.
- **Use Cases/Considerations:**

    - **Use When:** Assessing the security of critical third-party services, especially those handling sensitive data.
    - **Pros:** Provides independent validation of vendor security claims, reducing the need for extensive in-house audits.
    - **Cons:** Reports are snapshots in time; the vendor might implement changes after the audit.

3.8.2.2 Software Assurance (Validation, Verification, Testing)

- **Core Concept:** The level of confidence that software is free from vulnerabilities, performs its intended functions securely, and meets its security requirements. For third-party software, this often involves the receiving organization performing its own checks.
- **Flow/Structure:**
    - **Validation:** Ensuring the software meets the *intended* security requirements (e.g., does it protect PII as expected?).
    - **Verification:** Checking that the software was built *correctly* according to its design (e.g., does it implement the specified encryption algorithm?).
    - **Testing:** Conducting independent security tests (e.g., vulnerability scans, penetration tests) on the acquired component or service.
- **Software Application Context:** The organization runs its own **DAST** (Dynamic Application Security Testing, Domain 7.7) against a newly integrated third-party API endpoint to validate its input validation and authentication controls, even if the vendor provided their own test results.
- **Use Cases/Considerations:**
    - **Pros:** Provides direct assurance and identifies flaws that vendor attestation might miss.
    - **Cons:** Can be expensive and time-consuming, potentially impacting deployment schedules.

3.8.3 Formal Agreements

Definition

Legal contracts that explicitly define the security responsibilities, performance expectations, and liability between the organization and third-party vendors.

3.8.3.1 Service-Level Agreement (SLA)

- **Core Concept:** A contract that defines the level of service a vendor will provide, including specific security-related metrics (e.g., uptime, incident response times for security breaches, patching frequency).

- **Software Application Context:** The SLA for a cloud hosting provider guarantees 99.999% uptime and a **Recovery Time Objective (RTO)** of less than 4 hours in case of a security-induced outage (Continuity, Domain 3.1).
- **Use Cases/Considerations:**
    - **Pros:** Provides clear, measurable targets for security performance; includes penalties for non-compliance.

3.8.3.2 Master Service Agreement (MSA) and Statement of Work (SOW)

- **Core Concept:** The **MSA** is a foundational contract establishing general terms for all services between parties. The **SOW** is a document attached to an MSA, detailing the specific work, deliverables, and security requirements for a particular project or service.
- **Software Application Context:** The SOW for a custom software development project explicitly states that the vendor must adhere to the **OWASP Top 10 secure coding practices** (Domain 5.1) and submit all code for **SAST scanning** (Domain 7.8) before delivery.
- **Use Cases/Considerations:**
    - **Pros:** Ensures security requirements are legally binding and defined at the project level.

3.8.3.3 Memorandum of Understanding (MOU)

- **Core Concept:** A non-binding agreement between two or more parties outlining their intentions to work together. It's often a preliminary step before a formal contract, detailing shared security goals or cooperation during an incident.
- **Software Application Context:** Two companies might sign an MOU outlining their commitment to share threat intelligence or collaborate on incident response plans.
- **Use Cases/Considerations:**
    - **Pros:** Establishes a basis for cooperation without the full legal complexity of a contract.

3.8.3.4 Business Associate Agreement (BAA)

- **Core Concept:** A legally required contract between a **Covered Entity** (e.g., healthcare provider) and a **Business Associate** (a vendor) when the vendor will create, receive, maintain, or transmit Protected Health Information (PHI) on behalf of the Covered Entity (HIPAA compliance, Domain 3.2).
- **Software Application Context:** A cloud vendor hosting a healthcare application must sign a BAA, explicitly detailing their responsibilities for securing PHI and their legal liability under **HIPAA**.
- **Use Cases/Considerations:**

- **Use When:** Mandatory for **HIPAA** compliance.
- **Pros:** Clearly defines shared and individual responsibilities for PHI, reducing legal and regulatory risk.

Domain 4 – Secure Software Architecture and Design – 15%

Domain 4.1 - Define the Security Architecture

Definition: The systematic process of integrating security principles and controls into the structural design and blueprint of the software system. It establishes the trust boundaries, component interactions, and overall risk posture of the application before implementation begins.

4.1.1 Secure Architecture and Design Patterns

- **Core Concept:** Using established, repeatable blueprints to solve recurring security problems at the architectural level, reinforcing **Defense in Depth** (Domain 1.2.3).
    - **Sherwood Applied Business Security Architecture (SABSA):** A framework for creating an **enterprise security architecture** that aligns security requirements with business goals from the highest strategic level down to the component level. It defines security requirements in a business context first.
    - **Security Chain of Responsibility:** A design pattern where a request passes through a defined, sequential chain of security handlers (e.g., input sanitizer -> authentication filter -> authorization check). If any handler fails, the request is terminated (**Fail Secure**).
    - **Federated Identity:** An architectural pattern that allows users to authenticate once using an external identity provider (IdP) and gain access to multiple, disparate applications without re-authenticating (implementing **Single Sign-On - SSO**). Protocols like **SAML** and **OAuth 2.0** are common implementation types.

4.1.2 Security Controls Identification and Prioritization

- **Core Concept:** Translating identified **Risks** (Domain 2.8) and **Security Requirements** (Domain 3.1) into specific, actionable controls (technical, administrative, or physical) and prioritizing them based on **Business Risk**.
- **Prioritization:** High-impact, high-likelihood risks (e.g., injection attacks) require the highest priority **preventive controls** (e.g., Parameterized Queries) to be implemented first. The priority should align with the **Annualized Loss Expectancy (ALE)** (Domain 2.8).

4.1.3 Distributed Computing

- **Core Concept:** Systems where application components are spread across multiple network locations. Security focus shifts to **securing communication channels** and managing **trust boundaries** between components.
- **Types:**

- **Client-Server:** Traditional model; focus on securing the server and encrypting the client-server channel (**TLS**).
- **Peer-to-Peer (P2P):** All nodes are equal; securing P2P often requires strong **mutual authentication** and trust management between all endpoints.
- **Message Queuing:** Components communicate asynchronously via a message broker; focus on securing the broker and encrypting messages at rest within the queue.
- **N-tier:** Layered architecture (e.g., Web -> App -> Database); requires strict **segmentation** (firewall rules) and **Defense in Depth** between tiers.

4.1.4 Service-Oriented Architecture (SOA) (e.g., Enterprise service bus, web services, microservices)

- **Core Concept:** Designing the application as a collection of smaller, independent services, enforcing internal trust management.
  - **Enterprise Service Bus (ESB):** Middleware that acts as a central router and orchestrator. **Security Context:** A **Single Point of Failure (SPOF)** for security; must enforce security policies (authentication, logging) centrally and manage trust for all connections.
  - **Web Services:** Services exposing functionality via standard protocols (REST/SOAP). **Security Context:** REST security relies on **strong authentication tokens** (JWT) and strict **input validation**. Often protected by **API Gateways** to centralize external access control.
  - **Microservices:** Highly independent services, often containerized. **Security Context:** Enforces a strict **Zero Trust** model internally. **Implementation Focus:** Use a **Service Mesh** to automate **mutual TLS (mTLS)** for service-to-service authentication and isolation (Domain 1.2.8).

4.1.5 Rich Internet Applications (RIA)

- **Core Concept:** Modern web applications (e.g., SPAs) performing extensive processing on the user's browser (client-side).
- **Security Challenges: Client-Side Exploits or Threats** (e.g., XSS) and risks from **Remote Code Execution (RCE)** if server-side code is flawed. Threats are exacerbated by **Constant Connectivity** (long-lived sessions).
- **Implementation Focus:** Use **Content Security Policy (CSP)** headers to limit resource loading; implement rigorous server-side **Output Encoding** (Domain 5.1); enforce strict API authorization checks.

4.1.6 Pervasive/Ubiquitous Computing (e.g., Internet of Things (IoT), wireless, location-based, Radio-Frequency Identification (RFID), Near Field Communication (NFC), sensor networks, mesh)

- **Core Concept:** Devices embedded in the environment (sensors, RFID) that are often **resource-constrained** (low CPU/memory).
- **Security Context:** Challenges include lack of patchability, physical tampering risks, and weak protocols. Security relies on **network segmentation** (to isolate devices), **lightweight cryptography**, and secure provisioning using certificates.

4.1.7 Embedded Software (e.g., secure boot, secure memory, secure update)

- **Core Concept:** Software built directly into the hardware (firmware, drivers). Security must be enforced at the initial boot and hardware level.
- **Types of Security Mechanisms:**
  - **Secure Boot:** Verifies the cryptographic signature of the bootloader/OS kernel before execution (**Integrity**).
  - **Secure Memory:** Isolates sensitive code and keys in protected memory regions.
  - **Secure Update:** A verified, integrity-checked process for updating firmware to prevent malicious implants.
- **Implementation Focus:** Utilize the **Trusted Platform Module (TPM)** or other hardware anchors of trust.

4.1.8 Cloud Architectures (e.g., Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS))

- **Core Concept:** Deploying software on external, shared infrastructure. Security is defined by the **Shared Responsibility Model**, which varies by service model:
  - **IaaS:** Organization manages everything from the OS up (highest responsibility).
  - **PaaS:** Organization manages application code and data.
  - **SaaS:** Organization manages only data/access (least responsibility).
- **Implementation Focus:** Defining and enforcing the separation of duties via **Cloud Security Posture Management (CSPM)** tools and strict **IAM policies**.

4.1.9 Mobile Applications (e.g., implicit data collection privacy)

- **Core Concept:** Applications running on client devices (phones/tablets) outside the enterprise's perimeter.

- **Security Challenges: Insecure Data Storage** on the device; **Reverse Engineering** of code logic; **Implicit Data Collection Privacy** (accessing location/contacts without contextual consent, violating Domain 3.4).
- **Implementation Focus:** Use hardware-backed keystores (Keychain, Keystore) for sensitive data; apply **code obfuscation**; enforce **Certificate Pinning** for communication.

4.1.10 Hardware Platform Concerns (e.g., side-channel mitigation, speculative execution mitigation, secure element, firmware, drivers)

- **Core Concept:** Security vulnerabilities or features inherent in the physical processor or motherboard architecture that software must either leverage or mitigate.
- **Mitigation Examples:**
  - **Side-Channel Mitigation:** Protecting against attacks (e.g., timing, power consumption analysis) that infer cryptographic secrets using specialized software techniques (e.g., constant-time algorithms).
  - **Speculative Execution Mitigation:** Software patches (e.g., for **Spectre/Meltdown**) that restrict CPU optimizations to prevent unauthorized data leakage from memory.
  - **Secure Element (SE):** A specialized, tamper-resistant microchip used to securely store cryptographic keys, isolated from the main CPU.

4.1.11 Cognitive Computing (e.g., artificial intelligence (AI), virtual reality, augmented reality)

- **Core Concept:** Systems that mimic human intelligence (AI/ML, VR/AR). Security must protect the intellectual property and integrity of its decision-making.
- **Security Challenges: Model Poisoning** (corrupting training data) and **Adversarial Examples** (inputs designed to trick the running model).
- **Implementation Focus:** Strong **access controls** and **data loss prevention (DLP)** on training data stores, and rigorous validation of input data during both training and production inference.

4.1.12 Industrial Internet of Things (IoT) (e.g., facility-related, automotive, robotics, medical devices, software-defined production processes)

- **Core Concept:** IoT applied to critical sectors where security failure has high physical safety or environmental impact.
- **Security Context:** Prioritization of **Availability** and **Safety** often outweighs **Confidentiality**. Architecture must ensure extreme **Resiliency** (Domain 1.2.4) and utilize secure, authenticated firmware/software updates.

Domain 4.2 - Perform Secure Interface Design

Definition: The process of designing all external and internal communication points of the software (APIs, management consoles, data flows) to enforce security policies, manage trust boundaries, and prevent data leakage or unauthorized access.

4.2.1 Security Management and Log Interfaces

Definition

Interfaces intended for administrative control, monitoring, and auditing must be rigorously secured, often requiring separation from the main application network.

4.2.1.1 Security Management Interfaces

- **Core Concept:** Interfaces used by administrators to configure, monitor, or control security functions (e.g., firewall rules, user access provisioning, encryption key management). These are high-value targets for attackers.
- **How Implemented:**
    - **Access Control:** Access must be restricted via **Multi-Factor Authentication (MFA)** and strong **Least Privilege** policies. Ideally, separate credentials from application credentials are used.
    - **Auditing:** All actions performed via the management interface must be logged immutably (**Accountability**, Domain 1.1.6).
    - **Principle Applied: Segregation of Duties (SoD)**—separating administrative security functions from normal user access and code deployment.
- **Use Cases/Considerations:**
    - **Pros:** Centralizes control over critical security functions.
    - **Cons:** Represents a high-risk attack surface; a compromise can lead to system-wide control loss.

4.2.1.2 Out-of-Band Management

- **Core Concept:** Using a logically or physically **separate, isolated network channel** for administrative and security control access, distinct from the network used for regular application traffic.
- **Flow/Structure:** Administrative traffic (e.g., SSH, RDP) bypasses public-facing interfaces and is only accessible via a dedicated, restricted Virtual Private Network (VPN) or management network segment.

- **Software Application Context:** A developer can only access the application server's configuration console via a jump box or VPN connection that is entirely separate from the public-facing application network.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for managing infrastructure and core security appliances.
    - **Pros:** Enforces **Defense in Depth** (Domain 1.2.3); prevents an attack on the public interface from automatically granting access to the control plane.

4.2.1.3 Log Interfaces

- **Core Concept:** The interface through which security, application, and audit logs are transmitted from the application to a centralized collector (e.g., a SIEM system).
- **How Implemented:**
    - **Integrity:** The interface must use protocols that ensure log data is not tampered with during transit (e.g., authenticated and encrypted channels).
    - **Availability:** Must be resilient to failure to ensure logging continues even if the application experiences high load.
    - **Immutability:** Logs should be written to a **Write Once, Read Many (WORM)** storage system immediately upon arrival at the SIEM.
- **Software Application Context:** The application must be configured to forward security events over a secure, encrypted channel (e.g., TLS-encrypted Syslog) to the SIEM, preventing log tampering (Nonrepudiation).

4.2.2 Upstream/Downstream Dependencies

Definition

Managing the security of data and control flow when the application interacts with other services it either relies on (upstream) or provides services to (downstream). This defines the **trust boundaries** in distributed architectures.

4.2.2.1 Trust Boundaries and Data Flow

- **Core Concept:** A logical boundary in the application architecture where the level of trust changes. Data crossing this boundary **must be validated and sanitized**.
- **Flow/Structure:** All data received from an upstream component (e.g., a microservice or external API) must be treated as **untrusted input**, even if the upstream component is internal to the organization (**Zero Trust Principle**).
- **Software Application Context:** A security review identifies the trust boundary between the public-facing API Gateway (untrusted) and the internal authentication service

(trusted). The authentication service must strictly validate all input received from the gateway.

4.2.2.2 Key and Data Sharing Between Apps

- **Core Concept:** Managing the secure exchange of secrets (API keys, tokens, session data) and sensitive PII between different applications.
- **How Implemented:**
  - **Secrets Management: Never** hardcode secrets. Use a centralized **Secrets Vault** (e.g., HashiCorp Vault, AWS Secrets Manager) for all inter-app credentials.
  - **Data Minimization:** Only share the absolute minimum amount of data required (**Least Privilege**). Instead of sending a full customer record, send a unique, authorized identifier.
  - **Tokens:** Use standards like **JSON Web Tokens (JWT)** for data sharing, relying on digital signatures to ensure **Integrity** and **Authenticity** of the data.
- **Use Cases/Considerations:**
  - **Pros:** Reduces the risk of credential leakage and minimizes the compliance scope for PII.

4.2.3 Protocol Design Choices (APIs)

Definition

Selecting, designing, and using communication protocols and APIs in a way that minimizes security risks inherent in the protocol structure.

4.2.3.1 Application Programming Interfaces (API) and Weaknesses

- **Core Concept:** APIs are the primary interfaces in modern distributed systems. Poor API design often leads to vulnerabilities not found in traditional web applications.
- **Common API Weaknesses:**
  - **Broken Object Level Authorization (BOLA):** Flaw where the API does not properly check if a user has access to a *specific record* they are requesting.
  - **Excessive Data Exposure:** The API returns too much sensitive data that the client doesn't need, even if it's not displayed in the UI.
  - **Lack of Resource Limiting:** The API allows excessive requests, leading to DoS risks.
- **Implementation Focus:** Strict adherence to the **OWASP API Security Top 10** throughout the design phase. Use an **API Gateway** to centralize authorization and rate limiting.

4.2.3.2 State and Models

- **Core Concept:** Designing the security model around how the interface manages **state** (the data held between requests) and the interaction model (e.g., RESTful vs. RPC).
- **State Management:**
    - **Stateless:** The server retains no session information. Used in REST, where security relies on cryptographically signed tokens (e.g., JWT) that contain all necessary identity information (Nonrepudiation/Integrity). This adheres to **Economy of Mechanism**.
    - **Stateful:** The server maintains session data (e.g., using server-side session IDs). Security relies on robust **session management** (e.g., short expiration, regeneration after privilege change, and protection against session fixation).
- **Use Cases/Considerations:**
    - **Pros:** Choosing stateless APIs simplifies scaling and adheres to security principles but requires robust token management. Stateful models are easier for complex workflows but increase server-side attack surface.
    - **Cons:** Inconsistent state management can lead to **broken authentication** flaws.

Domain 4.3 - Evaluate and Select Reusable Technologies

Definition: The process of performing due diligence and making architectural choices to incorporate mature, secure, and commercially available technologies and components (software, hardware, or service) into the application design, rather than building custom, unvetted solutions.

4.3.1 Credential Management

Definition

Systems and practices used to securely handle the lifecycle of identities, secrets, and verification artifacts required for authentication and authorization.

4.3.1.1 X.509 (Certificates)

- **Core Concept:** A widely adopted standard defining the format for **Public Key Infrastructure (PKI)** digital certificates. Certificates bind a cryptographic public key to an identity (user, server, application) and are essential for securing TLS/SSL communication and verifying component authenticity.
- **How Implemented:** Used extensively for **mutual TLS (mTLS)** in microservices architectures and for **code signing** (Domain 5.6) to ensure integrity and origin.

- **Pros/Cons: Pros:** Provides strong, verifiable, non-repudiable identity assurance; enables mTLS for machine-to-machine communication. **Cons:** Requires complex **key and certificate lifecycle management** (issuance, renewal, revocation).

4.3.1.2 Single Sign-On (SSO)

- **Core Concept:** An authentication scheme that allows a user to log in with a single ID and password to gain access to multiple related, yet independent, software systems. This implements **Federated Identity** (Domain 4.1.1.1).
- **Types:** Uses standards like **SAML, OAuth 2.0, or OpenID Connect (OIDC)**.
- **Use Cases/Considerations:**
  - **Use When:** Integrating multiple internal applications or using external third-party cloud services.
  - **Pros:** Improves **Psychological Acceptability** (Domain 1.2.9); centralizes identity management and reduces the risk of password reuse across applications.
  - **Cons:** Creates a **Single Point of Failure (SPOF)** for authentication; if the SSO provider is compromised, all connected applications are immediately vulnerable.

4.3.2 Flow Control

Definition

Architectural components and protocols used to manage, inspect, and direct network traffic, isolating components and enforcing security policies at the network boundary.

4.3.2.1 Proxies (Forward and Reverse)

- **Core Concept:** A server that acts as an intermediary for requests from clients seeking resources from other servers. Proxies are essential for security and monitoring.
- **Types and Usage:**
  - **Forward Proxy:** Intermediary for **internal clients** seeking external resources. *Usage:* Enforces egress filtering (control what traffic leaves the network), blocks access to malicious sites, and caches external content.
  - **Reverse Proxy (API Gateway/WAF):** Intermediary for **external clients** seeking internal resources (e.g., web servers). *Usage:* Hides the internal network architecture, centralizes TLS termination, handles load balancing, and provides a layer of defense against web attacks (**Web Application Firewall - WAF**).
- **Examples:** A **Reverse Proxy** inspects all inbound HTTP requests for known attack signatures (SQLi, XSS) before forwarding traffic to the application server.

- **Pros/Cons: Pros:** Provides a critical **Defense in Depth** layer; centralizes logging and traffic shaping. **Cons:** Can introduce performance latency; requires specialized configuration and maintenance.

4.3.2.2 Firewalls

- **Core Concept:** Network devices or software that enforce access control policies between different security zones (trust boundaries) by monitoring and controlling incoming/outgoing network traffic.
- **Types: Network Firewalls** (traditional, focus on IP/Port), **Application Firewalls** (Web Application Firewalls - WAFs, focus on HTTP/S application layer), and **Host-Based Firewalls** (software on the server OS).
- **Software Application Context:** Used to enforce **network segmentation** (Domain 1.2.8) and restrict database access to only the specific application server IP address (**Least Privilege**).

4.3.2.3 Protocols

- **Core Concept:** Vetting and selecting secure communication protocols (e.g., TLS/SSL, SSH, IPsec) and disabling old, vulnerable protocols (e.g., SSL 3.0, TLS 1.0).
- **How Implemented:** Enforcing a policy that all inter-service communication (APIs) must use **TLS 1.3** and must disable insecure cipher suites.

4.3.2.4 Queuing

- **Core Concept:** Using message brokers (queues) to handle asynchronous communication between services. This is a flow control mechanism used to manage high load and decouple services.
- **Security Context:** Queues must implement controls to secure messages at rest (encryption) and enforce strong authentication/authorization for producers and consumers to prevent message injection.

4.3.3 Data Loss Prevention (DLP)

Definition

A set of tools and policies designed to detect and prevent the unauthorized extraction or disclosure of sensitive information (e.g., PII, credit card numbers, intellectual property) from the defined security boundaries.

- **How Implemented:** DLP systems scan data streams (**in motion**) and storage locations (**at rest**) for patterns matching sensitive data and automatically block, quarantine, or encrypt the data according to policy.

- **Software Application Context:** A DLP tool monitors the log interface (Domain 4.2.1.3) to ensure the application is **not logging unmasked PII** (violating confidentiality) and blocks any unauthorized external transfer of source code.
- **Use Cases/Considerations:**
    - **Pros:** Highly effective for compliance (PCI, HIPAA) by preventing accidental or malicious leakage of sensitive data.
    - **Cons:** Can generate significant **false positives** if not accurately configured, potentially disrupting business operations.

4.3.4 Virtualization

Definition

Technologies that create virtual versions of resources (hardware, OS, storage). Security assessment must focus on ensuring isolation between virtual guests.

4.3.4.1 Infrastructure as Code (IaC)

- **Core Concept:** Managing and provisioning infrastructure (networks, VMs, containers) through code (e.g., Terraform, Ansible) rather than manual processes.
- **Security Context:** IaC enforces a **secure baseline configuration** (**Configuration Removal**, Domain 2.6) every time resources are provisioned, preventing **configuration drift** and human error. Security reviews can be performed directly on the code (e.g., static analysis on IaC templates).

4.3.4.2 Hypervisor and Containers

- **Core Concept:** The **Hypervisor** manages the host machine and ensures secure isolation between virtual machines (VMs). **Containers** (e.g., Docker, Kubernetes) provide lightweight process-level virtualization.
- **Security Context:** The Hypervisor is the most critical component (**Trusted Computing Base - TCB**, Domain 4.3.5); securing the hypervisor is paramount. Containers require strict **resource limits** and the use of minimal, hardened base images to enforce **Least Common Mechanism** (Domain 1.2.8).

4.3.5 Trusted Computing

Definition

Hardware-based security technologies that establish a root of trust, ensuring the integrity and confidentiality of the computing environment from the moment it boots.

4.3.5.1 Trusted Platform Module (TPM)

- **Core Concept:** A dedicated cryptographic processor chip on a computer's motherboard used to securely store encryption keys, digital certificates, and platform measurement data. It provides a hardware root of trust.
- **How Implemented:** Used to enable **Secure Boot** (Domain 4.1.7) by storing cryptographic hashes of the kernel/firmware, verifying **Integrity** at startup, and providing keys for full-disk encryption.

4.3.5.2 Trusted Computing Base (TCB)

- **Core Concept:** The **sum total of all hardware, software, and firmware components** of a system that are critical to its security enforcement. A flaw in any component of the TCB could compromise the entire system.
- **Security Context:** The goal of secure design is to minimize the size and complexity of the TCB (**Economy of Mechanism**), making it easier to verify and audit. The OS kernel and hypervisor are typically part of the TCB.

4.3.6 Database Security

Definition

Techniques and tools used to protect data stored within the database and control access to the database engine itself.

- **Encryption: Field-level** (encrypting specific sensitive columns, e.g., SSNs) or **Disk/Volume-level** (encrypting the entire storage device).
- **Triggers:** Database mechanisms that execute code automatically in response to certain events (e.g., INSERT, UPDATE, DELETE). *Security Use:* Used to enforce security policies like **auditing** (automatically logging changes) or **integrity checks**.
- **Views:** Virtual tables based on the result-set of a query. *Security Use:* Used to enforce **Least Privilege** by restricting user access to a subset of columns or rows, masking sensitive data from unauthorized users.
- **Privilege Management:** Rigorously defining and limiting database user permissions (**Least Privilege**), often using **Role-Based Access Control (RBAC)** to manage access to tables, views, and stored procedures.
- **Secure Connections:** Enforcing the use of encrypted protocols (TLS/SSL) for all connections to the database server.

4.3.7 Programming Language Environment

Definition

The platform or environment in which the software is compiled and executed, which can provide built-in security features and memory protection.

- **Common Language Runtime (CLR) / Java Virtual Machine (VM):** Managed execution environments that offer automatic memory management (garbage collection), reducing the risk of memory corruption vulnerabilities (e.g., buffer overflows) common in native languages like C/C++. They also enforce sandbox or isolation models.
- **Python/PowerShell:** High-level scripting environments that reduce direct exposure to hardware/memory vulnerabilities but introduce risks related to insecure file I/O or reliance on external modules (supply chain risk).

4.3.8 Operating System (OS) Controls and Services

Definition

Leveraging built-in OS features to enforce security policies, including process isolation, access control, and memory protection.

- **Process Isolation:** The OS enforces memory separation between running applications to prevent one process from reading/writing the memory of another (**Least Common Mechanism**).
- **Access Controls (DAC/MAC):** Using Discretionary Access Control (DAC) or Mandatory Access Control (MAC) mechanisms at the OS level to enforce file and system resource permissions.
- **Security Enhanced Linux (SELinux) / AppArmor:** Tools that enforce mandatory access controls on Linux to restrict what processes can do, even if running as a privileged user.

4.3.9 Secure Backup and Restoration Planning

Definition

Ensuring the **Integrity** and **Availability** of data by implementing secure processes for data recovery.

- **Integrity:** Backups must be protected from ransomware and malware (often by **immutable storage**); a clean copy must be verified and cryptographically signed.
- **Restoration:** The plan must ensure that the process for restoring data is documented and tested frequently to meet the required **Recovery Time Objective (RTO)** and **Recovery Point Objective (RPO)** (Domain 3.1).
- **Encryption:** Backups must be **encrypted at rest** to maintain **Confidentiality**.

4.3.10 Secure Data Retention, Retrieval, and Destruction

Definition

Establishing reusable processes and tools that adhere to data disposition policies (Domain 2.6).

- **Retention:** Utilizing storage systems that can enforce **legal holds** and **WORM (Write Once, Read Many)** policies to comply with regulatory retention periods.
- **Retrieval:** Ensuring only authorized personnel can query and retrieve archived data (enforcing **Least Privilege**).
- **Destruction:** Employing validated tools for **cryptographic erasure** or secure overwriting when data reaches its End of Life (EOL).

Domain 4.4 - Perform Threat Modeling

Definition: A structured approach to identifying, prioritizing, and mitigating potential threats to a system. It's a proactive security practice performed early in the **Software Development Life Cycle (SDLC)** (Shift Left) to identify design-level flaws before coding begins, making it cost-effective and highly impactful.

4.4.1 Threat Modeling Methodologies

Definition

Systematic frameworks that guide the process of identifying threats, assessing their impact, and designing countermeasures. The choice of methodology depends on the project's scope, resources, and depth required.

4.4.1.1 Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE)

- **Core Concept:** A widely used, attacker-centric methodology developed by Microsoft that categorizes threats based on the security properties they violate. STRIDE provides a mnemonic to ensure a broad range of threats are considered.
- **How Implemented:**
    1. **Decompose the Application:** Break down the system into its components (data flows, data stores, processes, external entities) using **Data Flow Diagrams (DFDs)**.
    2. **Identify Trust Boundaries:** Mark where the level of trust changes (e.g., between the web server and database).
    3. **Apply STRIDE:** For each component and trust boundary, systematically ask "How could this be Spoofed?", "How could this be Tampered with?", etc.

4. **Mitigate Threats:** Design countermeasures for each identified threat.

- **STRIDE Categories:**
    - **Spoofing:** Impersonating something or someone else (e.g., identity spoofing, IP spoofing). *Violates Authentication.*
    - **Tampering:** Modifying data or code (e.g., SQL injection changing database records, altering firmware). *Violates Integrity.*
    - **Repudiation:** The ability of an attacker to deny performing an action (e.g., denying sending a message, denying a transaction). *Violates Non-repudiation.*
    - **Information Disclosure:** Exposing data to unauthorized individuals (e.g., leaking PII, stack traces). *Violates Confidentiality.*
    - **Denial of Service (DoS):** Preventing legitimate users from accessing a service (e.g., flooding a server, resource exhaustion). *Violates Availability.*
    - **Elevation of Privilege (EoP):** Gaining higher access rights than authorized (e.g., normal user becoming admin). *Violates Authorization.*
- **Pros/Cons: Pros:** Easy to learn and apply; comprehensive coverage of threat types; aligns directly with fundamental security properties. **Cons:** Can be tedious for very large systems; doesn't explicitly prioritize threats by risk likelihood (requires a separate step).

4.4.1.2 Process for Attack Simulation and Threat Analysis (PASTA)

- **Core Concept:** A seven-step, risk-centric methodology that integrates business objectives with technical requirements. PASTA focuses on attacker perspective and uses a step-by-step approach to identify vulnerabilities and define countermeasures. It’s more detailed and business-driven than STRIDE.
- **PASTA Steps:**
    1. **Define Objectives:** Business and security objectives for the application.
    2. **Define Technical Scope:** Components, technologies, data flows.
    3. **Decompose Application:** Create diagrams (DFDs, UML).
    4. **Analyze Threats:** Use threat libraries (e.g., OWASP Top 10) to identify threats relevant to the decomposed application.
    5. **Analyze Vulnerabilities:** Map identified threats to specific vulnerabilities in the architecture or code.
    6. **Attack Enumeration and Simulation:** Model potential attack paths.
    7. **Risk Analysis and Countermeasure Selection:** Prioritize risks based on likelihood and impact; select and design mitigating controls.

- **Pros/Cons: Pros:** Integrates business context; provides a comprehensive risk analysis; maps directly to technical vulnerabilities and countermeasures. **Cons:** More complex and time-consuming than STRIDE; requires expertise in risk management.

4.4.1.3 Hybrid Threat Modeling Methods

- **Core Concept:** Combining elements from different methodologies to create a tailored approach that fits specific project needs or organizational culture.
- **Examples:** Using DFDs and STRIDE for initial threat identification, then leveraging a risk scoring system (like CVSS) to prioritize, and finally using Attack Trees to simulate attack paths.
- **Pros/Cons: Pros:** Flexible and adaptable; leverages the strengths of multiple methodologies. **Cons:** Requires a good understanding of each component methodology; can become less standardized if not properly documented.

4.4.1.4 Common Vulnerability Scoring System (CVSS)

- **Core Concept:** An open, industry-standard framework for assigning a severity score to computer system vulnerabilities. It quantifies the characteristics of a vulnerability and provides a numerical score reflecting its severity.
- **Usage in Threat Modeling:** While not a threat modeling *methodology* itself, CVSS is used within methodologies (like PASTA) to **prioritize** identified threats and vulnerabilities based on their calculated severity.
- **Score Components:**
    - **Base Score:** Represents the intrinsic characteristics of a vulnerability (e.g., Exploitability, Impact).
    - **Temporal Score:** Reflects characteristics that change over time (e.g., availability of exploit code).
    - **Environmental Score:** Customizes the base/temporal scores based on the specific importance of the affected component within the organization.
- **Pros/Cons: Pros:** Standardized, objective, and widely accepted for vulnerability severity; helps prioritize remediation efforts. **Cons:** Can be subjective in assigning base metrics; does not account for specific threat actor capabilities.

4.4.2 Common Threats

Definition

Recurring categories of malicious actions or actors that commonly target software systems. Threat modeling must consider these archetypes.

4.4.2.1 Advanced Persistent Threat (APT)

- **Core Concept:** A sophisticated, often nation-state-sponsored or highly organized criminal group that gains unauthorized access to a computer network and remains undetected for an extended period. APTs typically target high-value information.
- **Characteristics:** Highly skilled, well-funded, patient, uses zero-day exploits, multi-stage attacks, custom malware, and extensive social engineering.
- **Threat Modeling Context:** When dealing with high-value targets (critical infrastructure, highly sensitive data), the threat model must assume a persistent, adaptive, and highly capable adversary, requiring a very robust **Defense in Depth** strategy.

4.4.2.2 Insider Threat

- **Core Concept:** A security risk that originates from within the targeted organization. This can be a current or former employee, contractor, or business partner who has legitimate access to systems.
- **Types:**
    - **Malicious Insider:** Deliberately uses their access to steal data, disrupt systems, or cause harm.
    - **Negligent Insider:** Accidentally causes harm through carelessness, lack of training, or bypassing security controls.
    - **Compromised Insider:** An attacker gains control of a legitimate insider's credentials.
- **Threat Modeling Context:** Requires a focus on **Least Privilege** (Domain 1.2.1), **Segregation of Duties (SoD)** (Domain 1.2.2), robust auditing (**Accountability**, Domain 1.1.6), and strong **data loss prevention (DLP)** (Domain 4.3.3). This directly relates to **Misuse Cases** (Domain 3.6.1.1).

4.4.2.3 Common Malware

- **Core Concept:** Malicious software designed to disrupt, damage, or gain unauthorized access to computer systems.
- **Types:** Viruses, worms, trojans, ransomware, spyware, rootkits.
- **Threat Modeling Context:** Considers how malware could enter the application's environment (e.g., through compromised third-party libraries - **Supply Chain Attack**, Domain 3.8.1.1), infect user systems (e.g., drive-by downloads via XSS), or leverage application vulnerabilities (e.g., insecure file uploads).

4.4.2.4 Third-Party Suppliers

- **Core Concept:** A threat originating from external vendors, partners, or open-source components that are integrated into the application (Software Supply Chain Risk, Domain 3.8.1.1).

- **Threat Examples:**
    - **Vulnerable Components:** Use of open-source libraries with known (CVEs) or unknown vulnerabilities.
    - **Compromised Supply Chain:** A vendor's build pipeline or update mechanism is infected (e.g., SolarWinds attack).
    - **Weak Vendor Security:** Third-party services with poor security controls (e.g., data breaches in SaaS providers).
- **Threat Modeling Context:** Requires thorough **vendor security assessments** (Domain 3.8), **Software Composition Analysis (SCA)** (Domain 7.9) for open-source dependencies, and robust agreements (SLAs, BAAs, Domain 3.8.3).

4.4.3 Attack Surface Evaluation

Definition

Identifying and analyzing all points where an unauthorized user can try to enter or extract data from a system. The goal is to reduce the attack surface.

- **Core Concept:** The sum of all code, inputs, outputs, services, interfaces, and open ports available to an attacker. **Minimizing the attack surface** is a fundamental security design principle (**Least Privilege**, Domain 1.2.1).
- **How Implemented:**
    1. **Map All Entry Points:** Document every API endpoint, web form, network port, exposed file share, and user interface element.
    2. **Identify All Data Inputs:** Every parameter, header, cookie, and file upload is a potential attack vector.
    3. **Analyze Exposed Functionality:** What operations can an unauthenticated or minimally privileged user perform?
    4. **Review External Dependencies:** What external services does the application call, and what data does it send them?
- **Tools/Techniques:**
    - **Data Flow Diagrams (DFDs):** Visually represent data inputs/outputs.
    - **API Documentation:** Lists all endpoints and parameters.
    - **Network Scanners:** Identify open ports.
- **Pros/Cons: Pros:** Provides a clear understanding of exposure; directly informs where to apply security controls. **Cons:** Difficult to keep current in highly dynamic environments (e.g., microservices, serverless).

4.4.4 Threat Analysis

Definition

The process of understanding and evaluating the identified threats, their potential impact, and their likelihood, leading to an informed decision about mitigation strategies.

- **Core Concept:** Moving beyond simply listing threats to understanding *which ones matter most* and *why*. It typically involves quantifying risk.
- **Steps/Components:**
    1. **Vulnerability Identification:** Map threats to specific vulnerabilities in the system (e.g., SQL Injection threat due to lack of parameterized queries).
    2. **Impact Assessment:** Determine the severity of damage if the threat is realized (e.g., financial loss, reputational damage, regulatory fines).
    3. **Likelihood Assessment:** Estimate the probability of the threat occurring, considering attacker capabilities, existence of exploits, and current controls.
    4. **Risk Prioritization:** Combine impact and likelihood to rank risks (e.g., High, Medium, Low).
- **Relationship to CVSS:** CVSS (Domain 4.4.1.4) provides a standardized way to quantify the *severity* of a vulnerability, which feeds into the impact assessment of a threat.
- **Outcome:** A prioritized list of risks that directly informs the design of **Mitigating Controls** (Domain 3.6.2).

4.4.5 Threat Intelligence

Definition

Knowledge about existing and emerging threats, including threat actors, their tactics, techniques, and procedures (TTPs), and indicators of compromise (IoCs).

- **Core Concept:** Proactive information that helps security teams anticipate and defend against attacks. Integrating threat intelligence into design means understanding the current threat landscape.
- **Types of Intelligence:**
    - **Strategic:** High-level information about threat actors and their motivations (e.g., "Nation-state group X targets financial institutions").
    - **Operational:** Information about specific TTPs used by threat actors (e.g., "APT group Y uses spear-phishing with Office macros").
    - **Tactical:** Technical data like **Indicators of Compromise (IoCs)** (e.g., malicious IP addresses, file hashes, malware signatures).

- **Usage in Threat Modeling:**
    - **Inform Threat Identification:** Helps identify relevant threats by understanding who might attack and how.
    - **Improve Likelihood Assessment:** Provides data on the prevalence and effectiveness of certain attack methods.
    - **Prioritize Controls:** Ensures that architectural decisions focus on defending against the most active and relevant threats.
- **Sources:** Commercial threat intelligence feeds, government advisories, open-source intelligence (OSINT), industry-specific sharing groups (e.g., ISACs).
- **Pros/Cons: Pros:** Proactive defense; helps tailor security efforts to real-world threats. **Cons:** Can be overwhelming without proper filtering; requires resources to consume and integrate effectively.

Domain 4.5 - Perform Architectural Risk Assessment and Design Reviews

Definition: Formal, systematic evaluation of the software architecture and design documentation before code implementation begins. The primary goal is to identify and mitigate high-impact, design-level flaws that are extremely expensive to fix later in the development lifecycle.

4.5.1 Architectural Risk Assessment

Definition

A methodology for identifying, analyzing, and quantifying security risks within the structural blueprint of an application. This builds upon the **Threat Modeling** activity (Domain 4.4).

4.5.1.1 Risk Identification and Analysis

- **Core Concept:** Using architectural diagrams (e.g., **Data Flow Diagrams - DFDs**) to systematically identify high-level risks before any code is written. This focuses on security issues inherent in the chosen technologies and component interactions.
- **Flow/Structure:**
    1. **Decompose the Architecture:** Break the system down into trust boundaries, data flows, and external interfaces.
    2. **Apply Threat Modeling:** Use methodologies like **STRIDE** (Domain 4.4.1.1) to brainstorm threats for each component.

3. **Analyze Risk:** Combine the likelihood of an attack occurring with the calculated **Business Impact** (Domain 2.8.4.1) to prioritize risks (e.g., Critical, High, Medium).

- **Software Application Context:** Identifying the risk of **Broken Access Control** across microservice boundaries due to a reliance on weak token validation, and assigning it a **Critical** risk score due to the PII exposure.
- **Use Cases/Considerations:**
  - **Pros:** Mitigation at this stage is inexpensive (changing a diagram is cheaper than rewriting code); focuses resources on the most impactful flaws (design defects vs. implementation bugs).
  - **Cons:** Requires specialized security expertise and deep knowledge of the proposed technology stack.

4.5.1.2 Mitigating Design Changes

- **Core Concept:** Implementing changes to the architecture to eliminate the risk entirely (**Risk Avoidance**) or significantly reduce its likelihood and/or impact (**Risk Mitigation**).
- **Implementation:**
  - **Risk Avoidance Example:** Deciding not to store PII locally, thus avoiding GDPR compliance risks entirely.
  - **Risk Mitigation Example:** Replacing a reliance on simple API keys for inter-service communication with **Mutual TLS (mTLS)** to reduce the likelihood of unauthorized access.
- **Principle Applied:** Enforcing **Defense in Depth** (Domain 1.2.3) by ensuring multiple layers of controls are present, especially at major trust boundaries.

4.5.2 Security Design Reviews

Definition

A formal, documented inspection of all security-relevant architectural and design documentation (**Security Architecture Document - SAD**, Domain 2.4.2.1) to confirm adherence to security policies and requirements.

4.5.2.1 Formal Review Process

- **Core Concept:** A mandated checkpoint (often a **Control Gate**, Domain 2.3.2.2) requiring a formal sign-off before proceeding to the implementation phase.
- **Participants:** The review typically involves the **Lead Architect**, **Security Team/Champion**, and sometimes the **Data Owner** (Domain 3.3.1.1).
- **Review Targets:**

    - **Requirements Coverage:** Ensuring all **Security Requirements** (Functional and Non-Functional, Domain 3.1) are addressed by the design (verified using the **SRTM**, Domain 3.7).
    - **Policy Compliance:** Verifying that the design aligns with organizational security policies (e.g., all authentication must use MFA).
    - **Component Vetting:** Checking that all reusable technologies (Domain 4.3) are approved, current, and securely configured by default.

4.5.2.2 Review Checklists and Security Principles

- **Core Concept:** Reviews are guided by standardized checklists that map directly to fundamental security design principles.
- **Checklist Examples:**
    - **Least Privilege:** Is every component/service assigned the absolute minimum permission necessary?
    - **Complete Mediation:** Is every single access attempt (not just initial login) checked for authorization?
    - **Isolation/Segmentation:** Are sensitive components (e.g., database) separated from public-facing components (enforcing **Least Common Mechanism**)?
    - **Fail Secure:** Does the system default to a deny state upon failure?

4.5.2.3 Deliverables and Documentation

- **Core Concept:** The design review must result in an official **Design Review Report** and formal updates to the project documentation.
- **Required Outputs:**
    - **Risk Acceptance/Mitigation Document:** Lists all identified risks and the approved treatment (e.g., accepted residual risk).
    - **Updated Security Architecture Document (SAD):** Reflects all required changes and additions of security controls.
    - **Sign-Off:** Formal authorization to proceed with coding (often linked to the **Assessment and Authorization - A&A** process, Domain 2.9.4).
- **Use Cases/Considerations:**
    - **Pros:** Creates **Accountability** for the design; guarantees security is **baked in**; essential for meeting regulatory requirements.
    - **Cons:** Can be perceived as a bottleneck if the process is not efficient or if the security team lacks the capacity to review designs promptly.

Domain 4.6 - Model (Non-Functional) Security Properties and Constraints

Definition: The process of defining the required quality attributes, constraints, and operational characteristics of a system's security features. These **Non-Functional Requirements (NFRs)** dictate the architecture's inherent toughness, resilience, and performance under security loads.

4.6.1 Security Properties and NFRs

Definition

Security NFRs are derived directly from the **Risk Assessment** (Domain 2.8) and compliance mandates (Domain 3.2). They are measurable goals that the final software must satisfy, often pertaining to the core CIA triad and its extensions.

4.6.1.1 Confidentiality and Integrity Constraints

- **Core Concept:** Requirements that specify the strength and rigor needed to protect data and prevent unauthorized modification.
- **Modeling Focus:** Defining acceptable cryptographic standards, hashing algorithms, and verification rigor.
  - **Cryptographic Strength:** Specifies the minimum acceptable key size and algorithm (e.g., "All data at rest must be encrypted with AES-256 using a FIPS 140-2 validated module").
  - **Integrity Assurance:** Defines the required level of data validation (e.g., "The system must use digital signatures for all inter-service communication to verify origin and data integrity").
- **Software Application Context:** The design must utilize a secrets manager that is certified to protect the encryption keys with an equivalent level of security as the data it protects.

4.6.1.2 Availability and Performance Constraints

- **Core Concept:** Requirements that define the security system's ability to maintain access during high demand and its performance impact on the overall application.
- **Modeling Focus:** Defining throughput, latency, and resilience goals.
  - **Security Performance:** Specifies the maximum acceptable latency introduced by security controls (e.g., "Multi-Factor Authentication (MFA) must complete within 2 seconds 99% of the time").

    - **Throughput:** Defines the rate at which security services must operate (e.g., "The authorization service must handle 10,000 authorization checks per second").

- **Use Cases/Considerations:** Performance NFRs are critical, as overly burdensome security controls often lead to users finding workarounds (**Psychological Acceptability**, Domain 1.2.9).

4.6.1.3 Resiliency and Continuity Constraints

- **Core Concept:** Requirements that define how the security system handles failures, ensuring it meets **Recovery Time Objectives (RTO)** and adheres to the **Fail Secure** principle.
- **Modeling Focus:** Defining redundancy and failover behavior.
    - **Failover/Redundancy:** Specifies the necessary architectural component duplication (e.g., "The centralized logging service must be deployed in an active-active cluster across two separate availability zones").
    - **Fail Secure Requirement:** Specifies the system's default state (e.g., "If the authorization service is unreachable, the system must **default to denying all access** and log the failure").

4.6.2 Modeling and Documentation

Definition

Translating abstract NFRs into concrete, verifiable models and documenting them in the **Security Architecture Document (SAD)**.

4.6.2.1 Quantitative and Qualitative Modeling

- **Core Concept:** Using formal methods to articulate NFRs precisely so they can be tested (Domain 6).
- **Quantitative Modeling:** Using measurable numbers (e.g., latency in milliseconds, percentage uptime, key length in bits). *Example: "Encryption key rotation must occur every 90 days."*
- **Qualitative Modeling:** Using descriptive statements tied to defined standards (e.g., "All secrets must be handled according to **PCI DSS 3.2**").
- **Flow/Structure:** These NFRs serve as the source requirements for the **Security Requirements Traceability Matrix (SRTM)** (Domain 3.7), ensuring they are verified during testing.

4.6.2.2 Mapping to Design Components

- **Core Concept:** Explicitly linking each security NFR to the specific architectural component responsible for implementing it.
- **Software Application Context:** The NFR "Achieve 99.999% availability" (Continuity) is mapped to the architectural decision to use an **Elastic Load Balancer** and **Geo-Redundant Database Replication**. This makes **Design Reviews** (Domain 4.5) more effective.
- **Use Cases/Considerations:**
    - **Pros:** Ensures that security constraints are built into the design from the start, avoiding costly post-implementation rework.
    - **Cons:** Poorly defined or overly ambitious NFRs can lead to "gold plating" or infeasible design choices.

Domain 4.7 - Define Secure Operational Architecture

Definition: The process of designing the operational environment and procedures (the "run-time" aspect of the architecture) to ensure the system remains secure and compliant from deployment through its end of life. This is the implementation of **DevSecOps** principles at the architectural level.

4.7.1 Deployment Topology and Isolation

Definition

The physical and logical layout of the running system, including network segmentation and environmental controls, designed to enforce **Defense in Depth** and limit attack scope.

4.7.1.1 Deployment Topology

- **Core Concept:** The specific arrangement of servers, containers, networks, and services that host the application. A secure topology enforces the **Principle of Least Privilege** (Domain 1.2.1) via segmentation.
- **How Implemented:** Defined using **Network Segmentation** (e.g., using different VLANs or subnets for the web tier, application tier, and database tier). The topology explicitly identifies all **Trust Boundaries** (Domain 4.2).
- **Software Application Context:** The database is placed in a private subnet (no public internet access), and access is restricted by a firewall to only the IP address of the application server (enforcing **Least Privilege** on network access).
- **Use Cases/Considerations:**
    - **Pros:** Limits the **blast radius** (Domain 1.2.8); an attacker breaching the web server cannot directly access the database without navigating the internal security controls.

4.7.1.2 Operational Interfaces

- **Core Concept:** Interfaces used for maintenance, monitoring, and scaling the running application (e.g., health check endpoints, configuration APIs). These must be secured as **management interfaces** (Domain 4.2.1).
- **How Implemented:** Operational interfaces should be isolated using **Out-of-Band Management** (Domain 4.2.1.2) networks or strict access control policies (e.g., only internal monitoring services can call the health check API).
- **Security Context:** Excessive logging (Domain 2.4) or unauthenticated debug interfaces must be disabled in production to avoid **Information Disclosure** (Domain 4.4.1.1).

4.7.2 Secure Continuous Integration and Delivery (CI/CD)

Definition

Integrating security tools and checks directly into the automated processes that build, test, and deploy software. This is the **DevSecOps** mandate applied to the operational pipeline.

4.7.2.1 Continuous Integration (CI) Security

- **Core Concept:** Ensuring that security checks are run automatically every time a developer commits code (integrating security into the build).
- **How Implemented:** The CI pipeline runs **Static Application Security Testing (SAST)** and **Software Composition Analysis (SCA)** on every code change. The results are used to enforce **Break/Build Criteria** (Domain 2.3.2.3), failing the build if critical flaws are found.
- **Security Context:** This ensures that security flaws are found and fixed in minutes or hours (**Shift Left**), dramatically reducing the **Average Remediation Time (ART)** (Domain 2.5).

4.7.2.2 Continuous Delivery (CD) Security

- **Core Concept:** Ensuring the security and integrity of the process that packages and deploys the verified application to the staging and production environments.
- **How Implemented:**
  - **Pipeline Integrity:** The CD system itself must be secured (e.g., using MFA, strong auditing, and **Least Privilege** for deployment accounts).
  - **Artifact Verification:** Before deployment, the system must verify the cryptographic signature or hash of the final software package (**Build Artifact Verification**, Domain 7.3) to confirm its **Integrity** and **Provenance** (Domain 8.3).

    - **Secure Provisioning:** Using **Infrastructure as Code (IaC)** to provision environment configurations securely every time, preventing **configuration drift** (Domain 7.5).
- **Use Cases/Considerations:**
    - **Pros:** Ensures that security flaws are not introduced during the deployment phase (e.g., accidental exposure of ports or insecure configuration).

4.7.3 Maintenance and Runtime Security

Definition

Designing the architecture to support secure patching, monitoring, and change management throughout the application's operational life.

4.7.3.1 Secure Change Management

- **Core Concept:** The architecture must support rapid, secure, and verifiable changes (e.g., patching, configuration updates) without compromising the system's security posture (Domain 2.9).
- **How Implemented:** Utilizing techniques like **blue/green deployments** or **canary releases** to implement changes with minimal risk. These strategies allow for continuous **Verification and Validation (V&V)** (Domain 2.9) of security controls in the new version before routing all user traffic.

4.7.3.2 Continuous Monitoring and Logging

- **Core Concept:** The application must be designed to generate sufficient, actionable security telemetry for ongoing threat detection.
- **Implementation:** The operational architecture includes a centralized **Security Information and Event Management (SIEM)** system. The application's logging interface (Domain 4.2) must be designed to output security-relevant events, such as authentication failures and authorization denials, in a standardized, correlated format for analysis (**Accountability**).
- **Use Cases/Considerations:**
    - **Pros:** Essential for detecting **Advanced Persistent Threats (APTs)** and validating that deployed security controls are actually working as intended.

Domain 5 – Secure Software Implementation – 14%

Domain 5.1 – Adhere to relevant secure coding practices

5.1.1 Declarative versus Imperative (Programmatic) Security

Definition

The choice of *where* security policies are defined and enforced within the application's structure.

Core Concept

This decision affects the separation of concerns, flexibility, and auditability of the security policy.

- **Declarative Security:** Policies are defined external to the business logic (e.g., annotations, configuration files, external policy engines). The runtime environment (e.g., J2EE container, Spring Security) intercepts the method call and enforces the rule automatically.
- **Imperative Security:** Policies are defined directly within the application code using conditional logic (if/else statements) surrounding the sensitive functions.

Implementation and Examples

- **Declarative Example:** Using a Java annotation like @RolesAllowed("admin") above a deleteUser() method. The security container handles the role check.
- **Imperative Example:** Inside the transferFunds() function, the developer writes an explicit if (user.getPermissions().contains("CAN_TRANSFER")) check.

Use Cases and Pros/Cons

Aspect	**Declarative Security**	**Imperative Security**
Use Case	Ideal for **standard authorization checks** (e.g., RBAC on entire modules).	Necessary for **complex, context-aware checks** (e.g., Row-Level Access Control, checking data ownership before action).
Advantage	**Separation of Concerns:** Policy is easily visible, auditable, and changeable without	**Flexibility:** Allows for complex logic based on real-time data or custom business rules.

Aspect	Declarative Security	Imperative Security
	recompilation.	
Disadvantage	Limited for fine-grained, data-dependent checks; often restricted by the framework's capabilities.	**High Risk:** Security logic is scattered, difficult to audit, and highly susceptible to developer error (e.g., missing an else block).

5.1.2 Concurrency (e.g., thread safety, database concurrency controls)

Definition

Managing security risks introduced when multiple processes or threads simultaneously access and manipulate shared resources, leading to unpredictable states or data corruption (**race conditions**).

Core Concept

Concurrency flaws can lead to severe security defects, such as two users updating an account balance simultaneously and corrupting the final value (**Integrity** violation) or one user overwriting another user's session token (**Authorization** bypass).

Implementation and Examples

- **Thread Safety (Code):** Ensuring code behaves correctly when executed simultaneously. *Example:* Using **Immutable Objects** (objects whose state cannot change after creation) or language-level primitives like synchronized blocks or locks (mutexes) on critical sections of code.
- **Database Concurrency Controls:** Mechanisms used by the database to manage simultaneous transactions to ensure **Integrity**. *Example:* Utilizing database **transactions** (ensuring ACID properties) or **optimistic locking** (checking a version number before committing an update).

Use Cases and Pros/Cons

Aspect	Thread Safety / Concurrency Controls
Use Case	Mandatory for financial transactions, session management logic, and any

Aspect	Thread Safety / Concurrency Controls
	cached shared data structures.
Advantage	Guarantees data **Integrity** and prevents **race conditions** that could lead to unauthorized privilege escalation or data corruption.
Disadvantage	Locks and synchronized methods can introduce **performance bottlenecks** (reduced **Availability**) and increase the risk of **deadlocks** if not implemented carefully.

5.1.3 Input Validation and Sanitization

Definition

The most critical defense against the majority of injection attacks (e.g., SQLi, XSS).

Core Concept

All input from untrusted sources (forms, APIs, cookies, file headers) must be treated as malicious until proven safe.

- **Validation:** Checking that input conforms to strict rules regarding **size, type, format, and content** before processing.
- **Sanitization:** Modifying input data to remove or neutralize malicious code, typically by escaping or converting dangerous characters.

Implementation and Examples

- **Allowlisting (Validation):** The *most secure* method. Defines what *is* allowed. *Example:* Rejecting all characters in a zipCode field except for 0-9 and the hyphen -. This eliminates the possibility of SQL injection.
- **Sanitization Example:** Removing JavaScript tags or converting the dangerous character < to the safe HTML entity &lt; before data is stored or processed.
- **Rule:** Validation must always be performed on the **server side** (never trust client-side validation), and should be performed as close to the input source as possible.

Use Cases and Pros/Cons

Aspect	**Input Validation and Sanitization**
Use Case	Mandatory for *every* data input accepted by the application.
Advantage	**Preventive Control** that stops up to 90% of common injection attacks; ensures data **Integrity**.
Disadvantage	Can be complex to implement correctly for rich, unstructured inputs (like formatted text fields or file uploads). Incorrect **denylisting** (blacklisting) is easily bypassed.

5.1.4 Error and Exception Handling

Definition

Securing how the application responds to unexpected failures (exceptions) to prevent information leakage while ensuring continuous service (**Resiliency**).

Core Concept

This practice is critical for adhering to the **Fail Secure** principle (Domain 1.2.4). Failure must result in the system moving to a safe state, not one that exposes sensitive data or grants unauthorized access.

Implementation and Examples

- **Do Not Leak Information:** All exceptions displayed to the end-user must be generic (e.g., "An unexpected error occurred"). *Avoid* displaying stack traces, database connection strings, internal file paths, or version numbers (**Information Disclosure**).
- **Log the Detail:** The full, technical exception details, including stack trace and contextual user/session data, must be logged *internally* for the security team to investigate (**Auditing**).
- **Fail Securely:** On critical errors (e.g., failure to connect to the authentication service), the system should halt the transaction and default to **denying all access**, rather than bypassing the security check.

Use Cases and Pros/Cons

Aspect	**Error and Exception Handling**
Use Case	Mandatory for every function call that involves access control, data handling, or external service communication.
Advantage	**Detective/Corrective Control** that prevents **Information Disclosure** to attackers; supports **Resiliency** by ensuring system recovery.
Disadvantage	Overly aggressive error handling can sometimes lead to **Denial of Service (DoS)** if the application shuts down too frequently or too broadly.

5.1.5 Output Sanitization (e.g., encoding, obfuscation)

Definition

Ensuring that data sent from the application back to the user's browser or another component is rendered safely, preventing the malicious code from executing.

Core Concept

This is the final line of defense against reflected or stored XSS attacks. The data is converted to a format safe for the interpreter (e.g., HTML, JavaScript).

- **Encoding:** Converting output data into a format safe for the intended context. *Example:* **HTML Entity Encoding** prevents a malicious <script> tag from being executed as code.
- **Obfuscation:** Deliberately obscuring the source code (e.g., mobile apps, client-side JavaScript) to make it difficult for an attacker to reverse engineer the business logic.

Implementation and Examples

- **Encoding Example:** Before displaying a user comment, the server applies HTML entity encoding to all special characters.
- **Obfuscation Example:** Running a JavaScript bundle through a minifier/obfuscator to rename variables and functions to unreadable names, protecting **Intellectual Property (IP)**.
- **Principle Applied: Complete Mediation** on data flow—the data is checked before being outputted.

Use Cases and Pros/Cons

Aspect	Output Sanitization and Encoding
Use Case	Mandatory for any data that is rendered to a user and originated from an untrusted source.
Advantage	**Preventive Control** that is the final safeguard against XSS; obfuscation protects **IP**.
Disadvantage	Obfuscation is *not* a security control; it's a deterrent (**Security by Obscurity**). Encoding must be context-specific (e.g., encoding for HTML is different from encoding for a JSON payload).

5.1.6 Secure Logging & Auditing (e.g., confidentiality, privacy)

Definition

Recording security-relevant events to provide an audit trail (**Accountability**) while ensuring that the log data itself remains confidential and compliant with privacy laws.

Core Concept

Logs are critical for **Detective** and **Corrective** controls (forensics, incident response). Poor logging can violate **Confidentiality** (by leaking secrets) or **Integrity** (by being tampered with).

Implementation and Examples

- **Confidentiality/Privacy: Never** log sensitive data (passwords, PII, session IDs, encryption keys) in plain text. PII must be **masked, hashed, or tokenized** before being written to the log file.
- **Integrity:** Logs must be protected from tampering (e.g., signed and forwarded quickly over a secure channel (TLS-encrypted) to a centralized, **immutable** SIEM system).
- **Event Logging:** Log *what* happened, *when*, *who* did it, and *the outcome* (success/failure). *Example:* Logging authorization failures, configuration changes, and successful logins.

Use Cases and Pros/Cons

Aspect	Secure Logging and Auditing

Aspect	**Secure Logging and Auditing**
Use Case	Mandatory for every security-relevant event (authentication, authorization, data modification).
Advantage	**Accountability** and **Nonrepudiation**; essential for **forensic analysis** and meeting regulatory compliance (e.g., HIPAA).
Disadvantage	Logging too much PII violates **Confidentiality/Privacy** laws; logs must be protected from resource exhaustion DoS attacks.

5.1.7 Session Management

Definition

Securely managing the lifecycle of a user's session from authentication to termination, maintaining state integrity, and preventing unauthorized session access.

Core Concept

A session token is the key to a user's access rights. Poor session management is a top security risk (e.g., **Broken Authentication**).

Implementation and Examples

- **Token Strength:** Sessions must use long, random, cryptographically strong, and unpredictable tokens (session IDs).
- **Regeneration: Regenerate the session ID** after any critical privilege change (e.g., after login, password change) to prevent **session fixation** attacks.
- **Expiration:** Implement short, reasonable expiration times (Absolute Timeout) and activity-based timeouts (Inactivity Timeout).
- **Protection:** Tokens should be transmitted only over **TLS/HTTPS** and stored securely (e.g., in **HTTPOnly cookies** to prevent XSS from accessing the token).
- **Example:** Forcing a user's session to expire immediately upon logging out or after 15 minutes of inactivity.

Use Cases and Pros/Cons

Aspect	Session Management
Use Case	Mandatory for any stateful application (web or API) that requires user login.
Advantage	Essential for enforcing **Authorization** throughout the user's interaction; prevents **session hijacking** and **session fixation**.
Disadvantage	Short session timeouts can be frustrating (**Psychological Acceptability**); maintaining state in stateless architectures (using tokens like JWT) requires strong cryptographic integrity checks.

5.1.8 Trusted/Untrusted Application Programming Interfaces (API), and Libraries

Definition

Managing security risks associated with the use of internal, external, and third-party code components (APIs and Libraries) within the application's environment.

Core Concept

All external components (third-party libraries, external APIs) introduce **Software Supply Chain Risk** (Domain 3.8). Even internal APIs must be treated with caution (**Zero Trust**).

Implementation and Examples

- **APIs:**
  - **Untrusted:** Treat all external APIs and data from client-side code as **Untrusted**. Subject them to the highest level of **Input Validation and Error Checking**.
  - **Trusted:** Internal APIs should still enforce **Least Privilege** and validate input (**Complete Mediation**) because the calling service might be compromised.
- **Libraries:**
  - **Vetting:** Use **Software Composition Analysis (SCA)** tools to identify known vulnerabilities (CVEs) in all dependencies.
  - **Secure Usage:** Use security libraries (e.g., an encryption library) as intended, avoiding insecure or custom configurations.

- **Principle Applied: Component Reuse** (Domain 1.2.10)—use pre-vetted, secure components, but manage the inherent risk.

Use Cases and Pros/Cons

Aspect	**Trusted/Untrusted APIs and Libraries**
Use Case	Mandatory whenever incorporating any external component (libraries, microservices, cloud APIs).
Advantage	**Component Reuse** speeds development; using vetted libraries reduces the risk of self-created security flaws.
Disadvantage	Introduces **Software Supply Chain Risk** (Domain 3.8); requires continuous monitoring (SCA) because vulnerabilities in external components are found daily.

5.1.9 Resource Management (e.g., compute, storage, network, memory management)

Definition

Securely controlling and limiting the application's consumption of system resources to prevent resource exhaustion attacks (DoS) and memory-based exploits.

Core Concept

This is a critical area for preventing **Denial-of-Service (DoS)** via resource exhaustion and **Remote Code Execution (RCE)** via memory flaws.

Implementation and Examples

- **Memory Management:** Preventing memory leaks and **buffer overflows**. Using managed languages (Java/Python) mitigates this risk by providing automatic memory clean-up.
- **Compute/Network:** Implementing **rate limiting** on APIs (e.g., limiting a user to 10 requests per second) and setting strict **timeouts** for long-running compute tasks.
- **Storage:** Securely releasing file handles after use to prevent resource locks; limiting file upload sizes and total storage quotas.

- **Example:** Setting a maximum content length on all incoming HTTP requests to prevent an attacker from uploading a massive file and consuming all disk space (Storage DoS).

Use Cases and Pros/Cons

Aspect	Resource Management
Use Case	Mandatory for all resource-intensive functions (file processing, database queries, bulk operations).
Advantage	**Preventive Control** against **Denial-of-Service (DoS)** attacks; prevents **RCE** by mitigating buffer overflows.
Disadvantage	Overly conservative resource limits can negatively impact legitimate users during peak loads.

5.1.10 Secure Configuration Management

Definition

The practice of establishing, documenting, and maintaining the **security posture** of the application and its environment throughout its lifecycle, preventing **configuration drift** from the approved secure baseline.

Core Concept

Insecure defaults and unintended configuration changes are common causes of security breaches. Configuration must be treated as code and subjected to version control and review.

Implementation and Examples

- **Baseline Security Configuration:** Defining a **hardened baseline** for the operating system (e.g., disabling unnecessary services), container image, and application server. This baseline should be enforced using **Infrastructure as Code (IaC)** (Domain 4.3) or configuration management tools (e.g., Ansible, Chef) to ensure consistency across Dev, Test, and Prod environments.
- **Credentials Management: Never** hardcode secrets (e.g., API keys, database passwords) into the source code or configuration files. *Example:* Retrieving secrets dynamically at runtime from a centralized, secure **Secrets Vault** (e.g., HashiCorp Vault, AWS Secrets

Manager). Secrets must be managed with **Least Privilege** and rotated regularly (e.g., every 90 days).

Use Cases and Pros/Cons

Aspect	**Secure Configuration Management**
Use Case	Mandatory for provisioning infrastructure and managing access keys in automated environments (CI/CD).
Advantage	**Consistency:** Ensures security controls are applied uniformly; reduces the risk of human error; vital for **Defense in Depth**.
Disadvantage	Initial setup and management of the IaC/Vault system is complex; requires a strict **Change Management** process (Domain 2.9) to prevent unauthorized changes.

5.1.11 Tokenization

Definition

The process of replacing sensitive data (e.g., credit card numbers, PII) with a non-sensitive equivalent, or **token**, that holds no extrinsic value.

Core Concept

This technique allows an application to process and store data for business purposes without ever possessing the original sensitive information, thereby **drastically reducing the scope of compliance** (e.g., for **PCI DSS** or **HIPAA**).

Implementation and Examples

- **Flow/Structure:** The application sends sensitive data to a trusted **Token Vault/Service** (which is subject to the strictest security controls). The service returns a non-sensitive token (e.g., 4523-XXXX-XXXX-1234). The application stores and processes the token instead of the original data.
- **Token Types:** Tokens can be **format-preserving** (matching the length/structure of the original data) or **non-format-preserving** (random strings).

Use Cases and Pros/Cons

Aspect	Tokenization
Use Case	Mandatory for payment processing, storing medical records, or any large volume of sensitive PII where compliance scope reduction is desired.
Advantage	**Compliance Reduction:** Data is no longer in scope for many regulations (since the app doesn't hold the keys); improves **Confidentiality** and minimizes the impact (**blast radius**) of a breach.
Disadvantage	Adds **latency** due to the communication with the token service; requires reliance on a dedicated, highly secure external system whose security is critical.

5.1.12 Isolation (e.g., sandboxing, virtualization, containerization, Separation Kernel Protection Profiles)

Definition

Architectural and coding techniques used to physically or logically separate system resources, processes, and data from one another to prevent a compromise in one component from affecting others (**Least Common Mechanism**, Domain 1.2.8).

Core Concept

Isolation is a fundamental architectural control ensuring that a security failure remains local, limiting the attacker's ability to move laterally (**Defense in Depth**).

Implementation and Examples

- **Sandboxing:** Running untrusted code (e.g., third-party plugins, user-uploaded scripts) in a highly restricted environment with minimal privileges and access to the host OS. *Example:* Web browsers running JavaScript in a sandbox to prevent malicious code from accessing the user's local file system.
- **Virtualization/Containerization:**
    - **Virtualization (VMs):** Provides strong **hardware-enforced isolation** using a hypervisor.

    - **Containerization (Docker/Kubernetes):** Provides lighter-weight **OS-level isolation** for processes. *Example:* Deploying each microservice in its own container with limited capabilities and user privileges.

- **Separation Kernel Protection Profiles (SKPP):** A type of high-assurance operating system that uses a formally verified **separation kernel** to provide strong, guaranteed isolation between different security domains (e.g., separating classified processing from unclassified storage).

Use Cases and Pros/Cons

Aspect	**Isolation**
Use Case	Mandatory for multi-tenant environments, running untrusted code, and high-assurance systems (SKPP).
Advantage	Essential for limiting the **blast radius**; allows components of different trust levels to run on the same hardware.
Disadvantage	Introduces complexity and performance overhead; container isolation requires careful hardening (e.g., proper user namespace and seccomp profiles).

5.1.13 Cryptography (e.g., payload, field level, transport, storage, agility, encryption, algorithm selection)

Definition

The implementation of mathematical principles and algorithms to protect data's **Confidentiality** (encryption) and **Integrity/Authenticity** (hashing, digital signatures).

Core Concept

Secure implementation requires both correct **algorithm selection** and correct **application** to the data's state (at rest, in transit, in use).

Implementation and Examples

- **Encryption Types by State:**

    - **Field Level Encryption:** Encrypting only specific, sensitive data elements (e.g., credit card numbers in a database column), allowing the rest of the database to be searchable.
    - **Payload Encryption:** Encrypting the entire body of a message or request (the payload) as it is processed by the application.
    - **Transport Encryption:** Encrypting data *in transit* across a network (e.g., using **TLS 1.3** for all communication).
    - **Storage Encryption:** Encrypting data *at rest* (e.g., using full disk or database volume encryption).
- **Cryptographic Agility:** The design goal of making it easy and quick to replace cryptographic algorithms (e.g., moving from RSA to elliptic curve) without massive code changes. This is necessary because algorithms inevitably become weak over time.
- **Algorithm Selection:** Choosing only standardized, publicly vetted, and currently strong algorithms (e.g., **AES-256**, FIPS-validated implementations) and avoiding proprietary or custom algorithms (**Open Design**, Domain 1.2.6).

Use Cases and Pros/Cons

Aspect	**Cryptography**
Use Case	Mandatory for protecting all data labeled "Confidential" or "Restricted" (Domain 3.3).
Advantage	Fundamental for achieving **Confidentiality** and **Integrity**; digital signatures provide **Nonrepudiation**.
Disadvantage	**Key Management is the Achilles' Heel:** Insecure storage or handling of encryption keys renders all cryptographic efforts useless. Introducing performance overhead (latency).

5.1.14 Access Control (e.g., trust zones, function permissions, role-based access control (RBAC), discretionary access control (DAC), mandatory access control (MAC))

Definition

Implementing the authorization logic (what a user or service can do) within the code, ensuring the **Least Privilege** principle is enforced at the function and data level.

Core Concept

Access control translates architectural boundaries and requirements into executable code that checks permissions before every resource request (**Complete Mediation**, Domain 1.2.6).

Implementation and Examples

- **Models:**
    - **Role-Based Access Control (RBAC):** Permissions are assigned to **roles** (e.g., "Auditor," "Developer"), and users are assigned to roles. *Example:* Checking if a user's role allows access to the admin_dashboard API endpoint.
    - **Discretionary Access Control (DAC):** The **owner** of a resource can grant or revoke access (e.g., sharing a document with specific colleagues).
    - **Mandatory Access Control (MAC):** System-enforced security labels (e.g., "Secret") control access, used in high-assurance environments where the system, not the user, dictates policy.
- **Trust Zones:** Access control rules are strictly enforced between defined security levels within the architecture (e.g., between the application server zone and the database zone).
- **Function Permissions:** Controlling access at the level of specific application functions or methods within the code (e.g., restricting who can call the initiatePayment() function).

Use Cases and Pros/Cons

Aspect	**Access Control Implementation**
Use Case	Mandatory for every function call or data access that involves sensitive information.
Advantage	Enforces **Authorization** and the **Least Privilege** principle; limits the damage caused by a compromised account.
Disadvantage	**Inconsistency:** Errors (e.g., missing a check) can lead to **Broken Access Control** (a top risk); MAC models are complex and rigid.

5.1.15 Processor Microarchitecture Security Extensions

Definition

Leveraging hardware features built into modern CPUs to provide enhanced security and isolation, reducing the effectiveness of low-level attacks against the operating system or application.

Core Concept

These extensions provide a **hardware root of trust** for low-level memory and execution integrity, protecting the most critical code from software attacks.

Implementation and Examples

- **Execute Disable (XD) / No-Execute (NX) Bit:** A processor feature used by the OS to mark memory regions as non-executable. *Example:* Prevents code injection attacks (e.g., **buffer overflows**) from executing malicious code placed in data buffers.
- **Intel SGX (Software Guard Extensions):** Creates secure, isolated memory regions (**enclaves**) within the CPU where sensitive code and data can be executed and stored, protected even from the OS kernel. *Example:* Running cryptographic key generation services inside an enclave to protect the keys from hypervisor-level attacks.
- **Mitigation:** The application code or the OS must be specifically configured (e.g., compiler flags, OS settings) to utilize these hardware capabilities.

Use Cases and Pros/Cons

Aspect	**Processor Security Extensions**
Use Case	Essential for high-assurance applications (e.g., payment processing, DRM, cloud key management) that require protection against low-level exploits.
Advantage	Provides strong, hardware-enforced **Confidentiality** and **Integrity** at the deepest level of the system.
Disadvantage	Requires specialized development (e.g., writing components to run in enclaves) and can introduce significant performance overhead.

Domain 5.2 - Analyse Code for Security Risks

5.2.1 Automated and Manual Code Analysis

5.2.1.1 Static Application Security Testing (SAST)

- **Core Concept:** A method for analysing the application's **source code**, byte code, or binary **without executing it**. SAST tools model data and control flows to identify common coding errors and security flaws (e.g., SQL Injection, buffer overflows). This is a **Preventive/Detective** control performed early in the SDLC (**Shift Left**).
- **How Implemented:** SAST is typically integrated into the **Continuous Integration (CI)** pipeline (Domain 4.7) or the developer's IDE.
  - **Automated Code Coverage:** Measures the percentage of the codebase analysed by the SAST tool to ensure comprehensive scanning.
  - **Linting:** A specific type of static analysis focused on code quality and style, often catching simple flaws like deprecated, insecure function calls.
- **Use Cases/Considerations:**
  - **Pros:** Finds flaws very early, making them cheap to fix; high code coverage.
  - **Cons:** High **False Positive Rate** (reporting non-issues); cannot detect configuration errors or runtime environment issues.

5.2.1.2 Manual Code Review (e.g., peer review)

- **Core Concept:** The manual, human-centric process of one or more security experts or peers inspecting code line-by-line to verify logic, adherence to coding standards (Domain 5.1), and security controls.
- **How Implemented:** Often done as a **Peer Review** during the pull request process. Reviews focus on high-risk areas identified by **Threat Modeling** (Domain 4.4), such as authentication, authorization, and cryptographic calls.
- **Use Cases/Considerations:**
  - **Pros: Excellent for finding logic flaws** and authorization defects (e.g., **Broken Object Level Authorization - BOLA**) that automated tools cannot model; provides a check on **compliance** with secure coding guidelines.
  - **Cons:** Time-consuming; effectiveness is limited by the reviewer's expertise; coverage is typically limited to critical modules.

5.2.2 Dependency and Integrity Checks

5.2.2.1 Secure Code Reuse

- **Core Concept:** The practice of utilizing previously developed, vetted, and approved code components or libraries (**Component Reuse**, Domain 1.2.10) to avoid reintroducing known vulnerabilities. This requires verifying the security of the reusable component itself.
- **Implementation:** Requires the use of **Software Composition Analysis (SCA)** tools (Domain 7.9) to audit all **open-source and third-party dependencies** against known vulnerability databases (**CVEs**).
- **Use Cases/Considerations:**
    - **Pros:** Reduces development time; safer than writing custom code for common functions (e.g., using a vetted crypto library).
    - **Cons:** Introduces **Software Supply Chain Risk** (Domain 3.8); requires continuous scanning to detect vulnerabilities found *after* the initial reuse.

5.2.2.2 Inspect for Malicious Code (e.g., backdoors, logic bombs, high entropy)

- **Core Concept:** The systematic search for intentional, malicious functionality, often inserted by insiders or compromised supply chain components, designed to evade standard testing.
- **Types of Malicious Code:**
    - **Backdoors:** Hidden methods or credentials that grant unauthorized remote access, bypassing normal authentication.
    - **Logic Bombs:** Code inserted to execute a malicious action (e.g., wipe a database) only when a specific, external condition is met.
    - **High Entropy:** Lack of expected randomness in code (e.g., a large block of base64-encoded or encrypted text that stands out) can flag unusual, possibly hidden, data blobs or executable code used for backdoors.
- **Detection Techniques:** Manual review of code changes by individuals with **Segregation of Duties (SoD)** is crucial to detect logic that violates the application's trust model.

5.2.3 External Vulnerability References

5.2.3.1 Vulnerability databases/lists

- **Core Concept:** Standardized external resources that define, categorize, and track known software security flaws. These lists guide developers and inform the configuration of SAST/DAST tools, ensuring efforts are focused on high-risk areas.

Open Web Application Security Project (OWASP) Top 10

- **Definition:** A standard awareness document and a high-priority checklist of the **top 10 most critical web application security risks**. It is used as a baseline for secure development training and penetration testing scope.

Rank	Category (Example)	Core Impact
A01:2021	**Broken Access Control**	**Authorization Bypass:** Flaws that allow an authenticated user to perform actions or access data they shouldn't (e.g., BOLA).
A02:2021	**Cryptographic Failures**	**Confidentiality Loss:** Flaws related to insecure data handling, key management, or outdated/weak encryption algorithms.
A03:2021	**Injection**	**Integrity/Control Loss:** Sending untrusted data to an interpreter (e.g., SQL, OS commands) to execute arbitrary commands.
A04:2021	**Insecure Design**	**Architectural Risk:** Flaws related to missing or ineffective control enforcement (e.g., missing threat modeling, poor separation of duties).
A05:2021	**Security Misconfiguration**	**Exposure:** Flaws due to unpatched systems, open ports, default accounts, or misconfigured security headers.
A06:2021	**Vulnerable and Outdated Components**	**Supply Chain Risk:** Using libraries or software components with known vulnerabilities (**CVEs**).
A07:2021	**Identification and Authentication Failures**	**Identity Compromise:** Flaws allowing poor password usage, weak MFA implementation, or broken session management.
A08:2021	**Software and Data Integrity Failures**	**Trust Loss:** Flaws related to unverified updates, insecure deserialization, or lack of integrity checks

Rank	Category (Example)	Core Impact
		on critical data.
A09:2021	**Security Logging and Monitoring Failures**	**Accountability Loss:** Flaws where security events are not logged, or logs are not monitored, preventing detection and forensics.
A10:2021	**Server-Side Request Forgery (SSRF)**	**Internal Access:** Flaws allowing an attacker to coerce the server into making arbitrary requests to internal network resources.

Common Weakness Enumerations (CWE)

- **Definition:** A dictionary of **software weaknesses** that provides a common language for describing flaws. Tools (SAST/DAST) use CWE IDs to tag findings.
- **Usage:** Used by security professionals and analysis tools to categorize, track, and standardize vulnerability remediation efforts, linking a high-level issue (Injection) to a specific weakness (CWE-89: Improper Neutralization of Special Elements).

SANS Top 25 Most Dangerous Software Errors

- **Definition:** A list of the most frequent and severe programming errors that lead to vulnerabilities, categorized by type (e.g., Insecure Interaction, Risky Resource Management). Often used for setting secure coding standards (Domain 5.1).

Domain 5.3 Implement Security Controls

Definition

The hands-on process of coding, configuring, and integrating specific technical safeguards into the application and its environment to enforce the security architecture (Domain 4) and mitigate identified risks (Domain 2.8). These controls are often **detective** or **corrective** in nature.

5.3.1 Watchdogs (Application Health Checks)

- **Core Concept:** A monitoring mechanism designed to detect when an application or a critical security function has failed or become unresponsive. If failure is detected, the watchdog can attempt remediation or trigger an alert. This is vital for maintaining **Availability** and **Resiliency**.

- **How Implemented:** Often implemented as a separate thread or external service that periodically pings a health check endpoint. If the endpoint fails to respond within a set timeout, the watchdog triggers a restart of the application process.
- **Software Application Context:** A watchdog monitors the **authentication service**. If the service stops responding, the watchdog restarts the service and triggers the application to enter a **Fail Secure** state (denying new logins) until the service is verified as fully operational.
- **Use Cases/Considerations:**
    - **Pros:** Automatically initiates recovery for simple failures, improving **Availability** (Domain 1.1.3).
    - **Cons:** Cannot fix complex logic errors and may mask deeper, chronic system issues if it restarts too frequently.

5.3.2 File Integrity Monitoring (FIM)

- **Core Concept:** A **detective control** that continuously monitors critical system, configuration, and binary files for unauthorized modifications, deletions, or additions.
- **How Implemented:** The system generates a cryptographic **hash** of all critical files (e.g., application binaries, web server configuration files, access control lists) and securely stores them (**Integrity** baseline). Periodically, the system recalculates the hash and compares it to the baseline.
- **Software Application Context:** FIM is used to monitor the web server's core configuration files (httpd.conf) and the application's deployed executable files. If an attacker breaches the server and attempts to plant a **backdoor** (malicious code, Domain 5.2.2.2) or modify the configuration, FIM detects the hash change instantly and alerts the security team.
- **Use Cases/Considerations:**
    - **Pros:** Provides high assurance of **Integrity** for deployed software; crucial for detecting unauthorized **rootkits** and malware.
    - **Cons:** Requires careful baseline management; frequent, legitimate file changes (e.g., log file rotation) can cause alert fatigue if not properly configured to ignore expected changes.

5.3.3 Anti-Malware (Anti-virus, Endpoint Detection and Response - EDR)

- **Core Concept: Preventive** and **Detective** software designed to identify, block, and remove malicious code (viruses, trojans, ransomware) that targets the host operating system or application environment.

- **How Implemented:** Scans files on access and at scheduled intervals using signature-based detection (known malware) and heuristic analysis (behavioral detection). Modern systems (EDR) also monitor process execution and system calls for suspicious activity.
- **Software Application Context:** Anti-malware runs on the application servers and build agents (CI/CD pipeline) to ensure that the deployment environment is not compromised, which could lead to a **Software Supply Chain** attack (Domain 3.8).
- **Use Cases/Considerations:**
    - **Pros:** Essential baseline defense against common, mass-market threats; helps secure the **Trusted Computing Base (TCB)** (Domain 4.3).
    - **Cons:** Often fails to detect sophisticated **Advanced Persistent Threats (APTs)** or **zero-day exploits**; requires constant signature updates.

Domain 5.4 Address the Identified Security Risks

Definition

The process of formally analyzing and deciding how to treat the risks identified during the security testing and code analysis phases (Domains 4.4 and 5.2). This involves selecting the appropriate **risk strategy** and documenting the decision.

5.4.1 Risk Strategy (Risk Treatment Options)

- **Core Concept:** Applying the final risk decision based on the **Risk Analysis** (Domain 2.8), weighing the business impact against the cost of mitigation.
- **Options and Application:**
    - **Mitigate (Reduce):** The most common option. Fixing the vulnerability by changing the code, implementing a new control, or patching the component. *Example: Fixing an XSS vulnerability by implementing **Output Encoding** (Domain 5.1).*
    - **Accept:** Formally acknowledging the existence of the risk and choosing to take no action. This is often done for low-risk findings where the cost of fixing far outweighs the **Annualized Loss Expectancy (ALE)**. This requires executive **Authorization** (Domain 2.9.4).
    - **Transfer:** Shifting the risk to another party. *Example: Purchasing cyber insurance to cover potential financial losses from a breach.*
    - **Avoid:** Removing the functionality or component that introduced the risk. *Example: Disabling a risky remote management protocol that is not essential for operations.*

- **Software Application Context:** A vulnerability is found in a third-party library. The team decides to **Mitigate** by updating the library, but if the patch is unavailable, they may choose to **Accept** the risk temporarily, provided the application is protected by a WAF.

5.4.2 Documentation and Accountability

- **Core Concept:** Ensuring that all risk treatment decisions are formally documented and tracked to maintain **Accountability** and facilitate future audits.
- **Documentation:** All decisions must be recorded in a **Risk Register** (Domain 2.8). This includes the original **Criticality Level** (CVSS score), the chosen treatment (Mitigate/Accept), the owner of the risk, and the final **Residual Risk** level.
- **Accountability:** For any risk that is **Accepted**, the decision must be reviewed and signed off by a relevant business owner or authorizing official (AO). This ensures that the risk is owned at the correct **business level** (Domain 2.8.4.1), not just by the development team.
- **Use Cases/Considerations:**
    - **Pros:** Ensures a data-driven approach to security spending; provides an auditable history of why a flaw was or was not fixed; critical for compliance.

Domain 5.5 Evaluate and Integrate Components

Definition

The process of vetting and integrating multiple software components, including external services and libraries, into a unified, secure application while managing the complex security dependencies introduced.

5.5.1 Systems-of-Systems Integration (SoS)

Core Concept

Integrating two or more independent, operational systems into a larger solution to achieve a common, complex goal. Security focus is on managing the **trust boundaries** between systems that were not designed to work together.

- **Trust Contracts:** Formal, legally binding or architectural agreements that explicitly define the security expectations and obligations between the two independent systems. *Example:* A contract might state that System A is responsible for authenticating the user, while System B is responsible for authorizing the user's specific action.
- **Security Testing:** Mandatory, focused testing to ensure secure operation across the interface. This includes **integration tests** that simulate cross-system traffic (e.g., verifying that data loss prevention (DLP) controls are effective on the interface).

- **Analysis:** Performing a joint **Threat Model** (Domain 4.4) and **Architectural Risk Assessment** (Domain 4.5) on the combined system to identify new, emergent risks that did not exist when the systems operated independently (e.g., cross-system attack chains).

Use Cases and Pros/Cons

Aspect	**Systems-of-Systems Integration**
Use Case	Merging two enterprise applications after a merger, or integrating an application with a major, legacy backend.
Pros	Enables powerful new business capabilities by combining existing functions; utilizes proven, existing components.
Cons	**High Risk:** Security failures in one system can cascade into the other; requires complex governance and dependency management.

5.5.2 Reusing Third-Party Code or Open-Source Libraries

Core Concept

The process of securely incorporating external, non-proprietary code into the application. This is a primary source of **Software Supply Chain Risk** (Domain 3.8).

- **Software Composition Analysis (SCA):** The automated process of scanning an application's codebase to identify all open-source and third-party components and libraries (including transitive dependencies). The tool then cross-references these components against vulnerability databases (**CVEs**) and license databases.
- **Secure Vetting:** Before integration, components must be vetted based on:
  - **Trustworthiness:** Is the library actively maintained and from a reputable source?
  - **Vulnerability Status:** Does it contain any known, unpatched CVEs?
  - **Licensing:** Is the license compatible with the proprietary application (Domain 3.8.3)?
- **Mitigation:** If a component contains a vulnerability, the team must **patch** the component (if possible) or wrap the component with **security controls** to prevent the flaw from being exploited (virtual patching).

Use Cases and Pros/Cons

Aspect	**Third-Party/Open-Source Reuse**
Use Case	Standard for modern development (e.g., using logging libraries, web frameworks).
Pros	**Component Reuse** (Domain 1.2.10) accelerates development; the code is often peer-reviewed and stable.
Cons	Introduces **Software Supply Chain Risk**; requires continuous monitoring (SCA) as new vulnerabilities are discovered daily.

Domain 5.6 Apply Security During the Build Process

Definition

Implementing security controls during the actual compilation and packaging of the software to protect the integrity of the executable and enforce secure standards. This is a key focus of **Continuous Integration (CI)** (Domain 4.7).

5.6.1 Anti-Tampering Techniques

- **Core Concept:** Methods used to deter attackers from modifying the deployed software (e.g., binaries, executables) and to detect if tampering has occurred.
- **Code Signing:** The use of a **Digital Signature** (Domain 1.1.2) applied to the final executable or deployment artifact. *Purpose:* Verifies the **Integrity** (the code hasn't been modified since signing) and the **Provenance** (who signed the code). The operating system or runtime environment can reject unsigned or tampered code.
- **Obfuscation:** Deliberately obscuring the code's logic (e.g., renaming variables, transforming code structure) to make **Reverse Engineering** difficult. *Note:* This is a **deterrent** and not a true security control (**Security by Obscurity**).

5.6.2 Compiler Switches

- **Core Concept:** Specific flags or options passed to the compiler during the build process that instruct it to enable or disable certain security protections in the resulting binary.
- **Implementation:** Compiler flags (e.g., in C/C++) can enable defenses against memory-based attacks. *Examples:*

    - **Stack Protection:** Enabling Canary or Stack Guard mechanisms to detect and prevent **buffer overflow** attacks.
    - **Address Space Layout Randomization (ASLR):** Configuring the binary to support ASLR, which randomizes memory addresses at runtime, making **Return-Oriented Programming (ROP)** attacks much harder to execute (Domain 7.11).
- **Use Cases/Considerations:**
    - **Pros:** Provides powerful, low-level **preventive controls** against memory corruption attacks.
    - **Cons:** Requires explicit configuration, as secure flags are not always enabled by default.

5.6.3 Address Compiler Warnings

- **Core Concept:** Treating compiler warnings—messages indicating potential problems in the source code—as potential security vulnerabilities that must be resolved before deployment.
- **Implementation:** Setting the compiler flag to treat all warnings as **errors**, which forces the build process to **break** (Fail) if any warnings are generated.
- **Security Context:** Many security-relevant vulnerabilities (e.g., implicit type casting, using deprecated functions, uninitialized variables) generate compiler warnings before they become exploitable. Resolving them enhances **code quality** and reduces the attack surface.
- **Principle Applied:** Enforcing **Break/Build Criteria** (Domain 2.3) based on code quality and standards violations.

Domain 6 – Secure Software Testing – 14%

Domain 6.1 - Develop Security Testing Strategy & Plan

6.1.1 Standards

Definition

Formal methodologies and compliance documents used as a blueprint for planning and executing comprehensive and repeatable security testing activities.

Core Concept

Using industry-recognized standards ensures the testing strategy is complete, effective, and aligns with organizational compliance goals.

Implementation and Examples

- **International Organization for Standardization (ISO):** ISO/IEC 27002, section 14.2.8 requires system security testing to be performed during development. *Example:* Adopting the **ISO/IEC 25010** standard for defining and measuring quality, including the security and reliability characteristics of the software.
- **Open-Source Security Testing Methodology Manual (OSSTMM):** A peer-reviewed manual providing a comprehensive framework for performing security testing and analysis. *Example:* Using OSSTMM's detailed modules to define the scope and metrics for a **Penetration Test** (e.g., defining the **Exposure Factor** and **Vulnerability Rating**).
- **Software Engineering Institute (SEI):** An organization focused on software engineering best practices. The SEI's contributions include guidance on secure development lifecycle management and risk mitigation strategies that inform the overall testing strategy.

Use Cases and Pros/Cons

Aspect	**Security Testing Standards (ISO, OSSTMM, SEI)**
Use Case	Establishing a test plan baseline for compliance purposes and for managing third-party security testing vendors.
Advantage	**Repeatability and Consistency:** Ensures tests are complete and results are comparable over time; provides external validation for the strategy.

Aspect	Security Testing Standards (ISO, OSSTMM, SEI)
Disadvantage	Standards are high-level and require significant effort to translate into specific, executable test cases.

6.1.2 Functional Security Testing (e.g., logic)

Definition

Testing the explicit, documented security features (**Functional Security Requirements**, Domain 3.1) to ensure they work exactly as intended and enforce the defined business logic.

Core Concept

Functional testing verifies that the positive and negative security features required by the business owner are correctly implemented and execute as designed.

Implementation and Examples

- **Positive Testing:** Testing that the security feature **grants access** when conditions are met. *Example:* Verifying that a user with the **"Admin" role** can successfully access the admin_dashboard interface.
- **Negative Testing (Logic):** Testing that the security feature **denies access** when conditions are *not* met, or when invalid/malicious input is provided. *Example:* Testing a **Misuse Case** (Domain 3.6) to ensure the system correctly enforces the limit of **three failed login attempts** before locking the account (logic).
- **Test Case Source:** Directly derived from the **Functional Security Requirements (FSRs)** and **Misuse/Abuse Cases** (Domain 3.6).

Use Cases and Pros/Cons

Aspect	Functional Security Testing
Use Case	Mandatory for verifying core features like authentication, authorization, input validation, and secure business rules.
Advantage	**Clear Pass/Fail:** Provides a clear, objective measure of compliance with FSRs;

Aspect	Functional Security Testing
	vital for finding flaws in authorization logic.
Disadvantage	Cannot test the resilience or performance of the system (that is non-functional testing).

6.1.3 Nonfunctional Security Testing (e.g., reliability, performance, scalability)

Definition

Testing the quality attributes and constraints of the security system, ensuring the application maintains **Confidentiality, Integrity, and Availability (CIA)** under various operational stresses.

Core Concept

Nonfunctional testing verifies the "how well" of security, ensuring security controls do not compromise the system's performance or resilience.

Implementation and Examples

- **Reliability:** Testing the application's ability to maintain defined security services under continuous operation. *Example:* Verifying that the security logging service (Domain 5.1.6) maintains **log integrity** without failure over a 72-hour test period.
- **Performance:** Testing the speed and efficiency of security controls. *Example:* Measuring the **latency** introduced by the **Multi-Factor Authentication (MFA)** process or the overhead added by **Transport Layer Security (TLS)** encryption (Domain 4.6).
- **Scalability:** Testing the ability of security controls (e.g., the authorization service) to handle increasing user load without degrading performance or failing. *Example:* Simulating 10,000 concurrent users to verify the **Session Management** system does not crash or allow session confusion.

Use Cases and Pros/Cons

Aspect	Nonfunctional Security Testing
Use Case	Essential for high-availability systems (e.g., financial trading platforms) and

Aspect	Nonfunctional Security Testing
	for verifying compliance with **Continuity Requirements** (Domain 3.1.2.3).
Advantage	Identifies bottlenecks and potential DoS risks (Availability failure); ensures security controls do not introduce undue performance burden.
Disadvantage	Requires specialized tools and dedicated test environments that accurately simulate production load and scale.

6.1.4 Testing Techniques (e.g., known environment testing, unknown environment testing, functional testing, acceptance testing)

Definition

The various methodologies and approaches used to define the perspective of the tester and the scope of the test.

Core Concept

Defining the testing technique (often called "testing perspective" or "knowledge level") determines the rigor and realism of the security assessment.

Implementation and Examples

- **Known Environment Testing (White Box):** The tester has **full knowledge** of the application's source code, architecture, internal documentation, and configuration. *Example:* Performing **Static Analysis (SAST)** or a manual **Peer Review** of the code (Domain 5.2). This is ideal for finding implementation-level flaws.
- **Unknown Environment Testing (Black Box):** The tester has **zero knowledge** of the internal workings of the application. The test is performed purely from an external, attacker's perspective. *Example:* Running a **Penetration Test** against the public API without prior access to the source code. This is ideal for finding perimeter vulnerabilities.
- **Functional Testing:** Focuses on verifying the security feature works (e.g., testing the **lockout mechanism**). (Same as 6.1.2).
- **Acceptance Testing:** The final phase of testing where the business owner or customer confirms that the system meets all contractual and **Security Requirements** (FSRs/NFRs) before formal deployment and **Authorization** (Domain 2.9.4).

Use Cases and Pros/Cons

Aspect	**Testing Techniques**
Use Case	White Box is essential for design-level assurance; Black Box provides the most realistic external attack simulation.
Advantage	Combining White and Black Box testing provides the strongest overall assurance (**Gray Box** testing is a mix of both).
Disadvantage	Black Box testing may not find deep, subtle flaws in the internal authorization logic or complex business rules.

6.1.5 Testing Environment (e.g., interoperability, test harness)

Definition

The specialized infrastructure and tools required to execute security tests accurately and consistently.

Core Concept

Security testing requires an environment that is as close to production as possible while being isolated from it, often necessitating dedicated tools to manage the process.

Implementation and Examples

- **Interoperability:** Ensuring the application's security components (e.g., SSO service, logging interface, microservices) work correctly and securely together within the integrated environment. *Example:* Verifying that the application can successfully authenticate against the **Federated Identity** service and maintain session state across services.
- **Test Harness:** A specialized framework, set of drivers, or tool that simulates the production environment and automates the execution of security test cases. *Example:* A test harness might simulate a high volume of unauthenticated requests to perform **load testing** on the authorization component.
- **Environment Isolation:** The test environment (staging/QA) must be completely isolated from the production network to prevent testing errors or malicious testing traffic from impacting the live system (**Defense in Depth**).

- **Test Data Security:** The testing environment must use **Secure Test Data** (Domain 6.7), often anonymized or synthetic data, instead of real PII from production (Domain 3.4).

Use Cases and Pros/Cons

Aspect	**Testing Environment**
Use Case	Mandatory for non-functional testing (load, performance) and final acceptance testing.
Advantage	Enables accurate simulation of production risks and performance without impacting real users; ensures **Interoperability** of security controls.
Disadvantage	Creating a dedicated environment that perfectly mirrors production is expensive and resource-intensive.

6.1.6 Security Researcher Outreach (e.g., bug bounties)

Definition

Engaging external, independent security experts and researchers to help identify vulnerabilities, leveraging external expertise and diverse perspectives.

Core Concept

Formalizing the relationship with the external security community to benefit from their specialized skills and provide a legal, compliant avenue for reporting vulnerabilities.

Implementation and Examples

- **Bug Bounties:** A formalized, public or private program that offers financial rewards to independent security researchers who responsibly discover and report valid vulnerabilities in the application.
- **Vulnerability Disclosure Program (VDP):** A public policy outlining how external researchers can legally and responsibly report vulnerabilities, even without offering a monetary reward. This provides **legal clarity** (Domain 2.8) and protects the organization from being blindsided by public disclosures.
- **Responsible Disclosure:** A practice where the researcher contacts the organization privately and gives them time to patch the flaw before disclosing it publicly.

Use Cases and Pros/Cons

Aspect	**Security Researcher Outreach**
Use Case	Ideal for high-profile, public-facing applications (e.g., e-commerce, cloud services) where external testing provides significant value.
Advantage	**Cost-Effective:** Pay for results only; leverages a large, diverse pool of external expertise.
Disadvantage	Requires dedicated internal resources to triage and manage the flood of incoming reports (including false positives); risk of **Information Disclosure** if the program is not managed tightly.

Domain 6.2 - Develop Security Test Cases

Definition: The systematic process of designing and executing specific methods (manual and automated) to verify and validate that the application's security controls are effective and that the application is free from exploitable security flaws.

6.2.1 Proactive Security Testing Techniques

6.2.1.1 Attack Surface Validation

- **Definition:** The process of identifying and mapping all possible entry and exit points where a user or data can interact with the system. **Validation** confirms that the deployed application has no exposed, undocumented interfaces or unnecessary open ports.
- **Core Concept:** A proactive step in the **Threat Modeling** process (Domain 4.4). The goal is to **minimize the attack surface** (Domain 1.2.1) by verifying that only necessary interfaces are accessible.
- **Implementation:** Using **network scanning tools** (e.g., Nmap) to confirm that firewall rules and deployment configurations correctly block traffic to internal services and management ports (Domain 4.2).

6.2.1.2 Misuse and Abuse Test Cases

- **Definition:** Formal, structured test cases derived from **Misuse and Abuse Cases** developed during the requirements phase (Domain 3.6).

- **Core Concept:** These tests validate the effectiveness of the **Mitigating Controls** (Domain 3.6) designed to prevent malicious behavior (**Abuse**) or unauthorized rule-bending by authenticated users (**Misuse**).
- **Implementation:** Test cases involve providing malicious input (e.g., SQL injection strings, oversized files) or attempting to bypass authorization rules (e.g., accessing another user's data record by manipulating an ID in the URL).

6.2.1.3 Cryptographic Validation

- **Definition:** Testing the correct implementation of cryptographic functions and verifying the quality of the underlying randomness used for keys and tokens.
- **Core Concept (Entropy): Entropy** is the measure of the **unpredictability or randomness** in a data source. Cryptographic strength is directly proportional to the quality of the entropy source used to generate secrets (keys, nonces, initialization vectors). Low entropy renders even strong algorithms weak because an attacker can easily guess the key.
- **Validation Focus:**
    - **Pseudorandom Number Generators (PRNGs):** Testing that PRNGs used for key and session generation are cryptographically secure and draw from a high-quality entropy pool (e.g., hardware-backed sources like /dev/random).
    - **Algorithm Selection:** Verifying that the application uses standardized, FIPS-validated, and currently strong algorithms (Domain 5.1).

6.2.2 Automated Vulnerability Testing

6.2.2.1 Dynamic Application Security Testing (DAST)

- **Definition:** Testing the security of a **running application** by feeding it input and observing its output and behavior from the **outside**. DAST simulates an attacker (**Black Box testing**).
- **Core Concept:** DAST identifies vulnerabilities that manifest only in the runtime environment (e.g., configuration errors, environmental flaws). It checks for issues like XSS, CSRF, and misconfigured HTTP headers.
- **Pros/Cons: Pros:** Finds runtime and configuration issues; provides clear proof-of-concept of exploitability. **Cons:** Cannot see source code; low code coverage compared to SAST; only finds flaws in code paths that are executed.

6.2.2.2 Interactive Application Security Testing (IAST)

- **Definition:** A hybrid approach that uses an agent deployed *inside* the running application to monitor execution flow, data flow, and libraries while external tests are run.

- **Core Concept:** IAST observes the inner workings of the application (like SAST) while running external tests (like DAST). This provides **higher accuracy** and better data on the precise location of the vulnerability in the source code.
- **Pros/Cons: Pros:** Lower false positive rate than SAST or DAST alone; provides **high confidence** in the vulnerability's location in the code; ideal for complex APIs. **Cons:** Requires integration with the application runtime; can introduce performance overhead in the test environment.

6.2.2.3 Penetration Tests

- **Definition:** A comprehensive, controlled simulation of a real-world attack performed by **human security experts** against the application and its environment.
- **Core Concept:** Pen tests validate whether security controls (e.g., authentication, firewalls) can be bypassed and find complex, multi-step **attack chains** that automated tools miss.
- **Types of Testing: Security Controls** bypass attempts, checking for **Known Vulnerabilities**, and simulating **Known Malware** introduction.
- **Pros/Cons: Pros:** Finds complex logic flaws; provides the most realistic assessment of **Business Risk**; critical for final **Authorization** (Domain 2.9). **Cons:** Expensive and time-consuming; limited scope based on the contract.

6.2.3 Stress and Integrity Testing

6.2.3.1 Fuzzing

- **Definition:** An automated testing technique that involves inputting large amounts of **malformed, unexpected, or random data** (**fuzz**) into a program to try and cause crashes, memory leaks, or application failures.
- **Core Concept:** Highly effective for finding bugs in input parsers and protocol handlers, which often lead to **buffer overflows** or **Denial of Service (DoS)**.
- **Types: Generated Fuzzing** (creating entirely new inputs based on models) or **Mutated Fuzzing** (slightly modifying known, valid inputs).

6.2.3.2 Failure (Fault Injection, Stress Testing, Break Testing)

- **Definition:** Testing the application's **Resilience** and **Availability** by deliberately inducing abnormal or failure conditions to ensure the system fails gracefully and securely.
- **Fault Injection:** Introducing specific, targeted errors (e.g., forcing an API call to return a 500 error, dropping network packets) to verify that the application handles the failure without compromising integrity or crashing (**Fail Secure principle**, Domain 1.2.4).

- **Stress Testing:** Pushing the system beyond its expected operational limits (e.g., overwhelming the authorization service) to verify that it degrades gracefully and does not expose security flaws under extreme load.

6.2.3.3 Simulation

- **Definition:** Using synthetic data and environments that accurately mirror the complexity and scale of the live system to make testing realistic.
- **Core Concept: Simulating production environment and production data** is essential for realistic performance and security testing, without using sensitive PII (Domain 3.4). **Synthetic transactions** (e.g., automated, repeatable scripts mimicking user behavior) are run against this simulated environment.

6.2.4 Lifecycle Testing Integration

6.2.4.4 Unit testing and code coverage

- **Definition:** Testing the smallest testable parts of an application (**units**) to ensure they work correctly. **Code coverage** measures the percentage of code executed by the tests.
- **Security Context:** Critical for checking that individual **security controls** (e.g., a password hashing function, an input validation utility) work exactly as intended (**Verification**).

6.2.4.5 Regression tests

- **Definition:** Re-running existing test cases (including security test cases) after a change to ensure that the changes did not introduce new defects or cause old, fixed defects to reappear.
- **Security Context:** Ensures that fixing one vulnerability did not accidentally break a previously working security control.

6.2.4.6 Integration tests

- **Definition:** Testing combined groups of components (e.g., microservices, database, authentication module) to ensure they work correctly together as a whole.
- **Security Context:** Critical for validating secure interactions across **Trust Boundaries** (Domain 4.2), especially authentication and authorization flows between services.

6.2.4.7 Continuous testing

- **Definition:** The practice of executing automated tests (SAST, DAST, Unit, Regression) continuously throughout the entire development pipeline.
- **Core Concept:** A key mandate of **DevSecOps** and the **Shift Left** approach, where security checks are run immediately upon every code commit.

Domain 6.3 Verify and Validate Documentation

Definition

The process of reviewing all user-facing and administrator-facing documentation to ensure it is accurate, complete, and does not unintentionally expose security flaws, sensitive information, or misuse of the application.

Core Concept

Documentation is a critical component of the **Trusted Computing Base (TCB)** (Domain 4.3.5); if it is wrong, users or administrators may operate the system insecurely, or attackers may gain valuable reconnaissance. Documentation must be *verified* for accuracy and *validated* for usability.

Implementation and Examples

- **Installation and Setup Instructions:** These must be reviewed to ensure they enforce a **Secure Baseline Configuration** (Domain 5.1.10) from the start. *Example:* Verifying that instructions mandate disabling insecure default accounts, using strong, non-default passwords, and restricting network access after installation.
- **Error Messages:** These must be validated to ensure they adhere to **Error Handling** rules (Domain 5.1.4). *Example:* Checking that error messages shown to the user are generic and do not expose internal details like stack traces, database schemas, or internal file paths (**Information Disclosure**, Domain 4.4.1.1).
- **User Guides and Administrator Guides:** Reviewed to ensure they correctly explain how to use security features (e.g., MFA, encryption key rotation) and do not provide instructions that facilitate a security bypass.
- **Release Notes:** Must be checked to ensure they only detail necessary fixes and features, and do not inadvertently reveal the existence of specific, unpatched vulnerabilities (which could inform attackers).

Use Cases and Pros/Cons

Aspect	**Documentation V&V**
Use Case	Mandatory before every major release to prevent accidental information leakage and ensure secure operation.

Aspect	Documentation V&V
Advantage	Prevents human error in deployment (due to poor instructions); prevents **Information Disclosure** to attackers.
Disadvantage	Often overlooked or prioritized too late, leaving a window for documentation-based security flaws.

Domain 6.4 Identify Undocumented Functionality

Definition

The systematic process of identifying features, APIs, debug interfaces, or developer backdoors that were implemented in the code but were never documented or intended for use by the end-user.

Core Concept

Undocumented functionality often bypasses standard security controls and presents a high-risk **Attack Surface** (Domain 4.4.3). This is a target for malicious insiders or external reverse engineering.

Implementation and Examples

- **Detection:** This activity is often performed during **Manual Code Review** (Domain 5.2.1.2) or **Penetration Testing** (Domain 6.2.2.3) by attempting unexpected inputs or running network scans for hidden services.
- **Interface Enumeration:** Checking the application's configuration files, binary strings, and network communication for hidden URLs, debug ports, or developer endpoints.
- **Risk:** Undocumented functions often lack proper **Authentication** and **Authorization** checks, making them potential **Backdoors** (Domain 5.2.2.2). *Example:* A hidden developer command-line interface (CLI) that was left active in the production binary, allowing unauthenticated configuration changes.

Use Cases and Pros/Cons

Aspect	Identify Undocumented Functionality

Aspect	**Identify Undocumented Functionality**
Use Case	Essential for mitigating **Insider Threat** (Domain 4.4.2.2) and ensuring the **Attack Surface** is minimal.
Advantage	Removes high-risk, unvetted interfaces that bypass the system's defined security model.
Disadvantage	Requires deep knowledge of the source code and architecture (often a white-box activity).

Domain 6.5 Analyse Security Implications of Test Results

Definition

The formal process of interpreting the findings from security testing (SAST, DAST, Penetration Tests) and translating the technical findings into actionable business impact and remediation strategy.

Core Concept

Test results must move beyond technical severity (e.g., CVSS score) to determine the **Business Risk** (Domain 2.8.4.1) and trigger the correct **risk strategy** (Domain 5.4.1).

Implementation and Examples

- **Impact on Product Management:**
    - **Prioritization:** Assigning remediation tasks based on a risk matrix that weighs **Technical Severity (CVSS score)** against **Business Impact** (e.g., PII exposure, compliance fines). A high-CVSS flaw in an internal, non-critical tool may be downgraded, while a medium-CVSS flaw in a customer-facing financial API is **Critical**.
    - **Resource Allocation:** Justifying the time and cost required to fix the vulnerability (Domain 2.8) and scheduling the fix into the product roadmap or Agile backlog.
- **Break/Build Criteria:**

    - Test results are used to refine and enforce automated release gates (**Control Gates**, Domain 2.3.2.2). *Example:* The discovery of a remote code execution (RCE) flaw during DAST leads to a new automated rule: "The build **must break** if *any* RCE pattern is detected by SAST/DAST."

- **Formal Documentation:** The analysis results are formally documented in the **Risk Register** (Domain 2.8) as input for the **Assessment and Authorization (A&A)** process (Domain 2.9.4). The remediation plan must track the **Average Remediation Time (ART)** (Domain 2.5).

Use Cases and Pros/Cons

Aspect	**Analyse Security Implications**
Use Case	Mandatory after every formal security assessment (Penetration Test, DAST).
Advantage	Ensures security efforts are focused on threats that pose the greatest **Business Risk**; provides clear, data-driven justification for development resource allocation.
Disadvantage	Requires effective communication between the technical security team and non-technical business stakeholders (e.g., Data Owners) to accurately define risk acceptance and priority.

Domain 6.6 Classify and Track Security Errors

Definition

The systematic process of analysing, quantifying, and prioritizing security flaws identified during testing, and managing their remediation through formal tracking systems.

6.6.1 Bug Tracking (e.g., defects, errors, and vulnerabilities)

- **Core Concept:** Using a formal **bug tracking system** (e.g., Jira, Azure DevOps) to log, assign ownership, manage the workflow status, and audit the history of every identified flaw. This ensures **Accountability** (Domain 1.1.6) for remediation.

- **Classification:** Flaws are categorized by their nature:

    - **Defects/Errors:** Failures to meet non-security functional requirements (e.g., a button doesn't work).
    - **Vulnerabilities:** Failures to meet security requirements that create an exploit path (e.g., a SQL injection flaw).

- **Implementation:** All identified **Vulnerabilities** must be logged as critical tickets with associated **Risk Scores** (Domain 6.6.2) and assigned an **Average Remediation Time (ART)** target (Domain 2.5). The tracking system is essential for maintaining the **Risk Register** (Domain 2.8).
- **Use Cases/Considerations:**
    - **Pros:** Ensures no security flaws are lost; provides an auditable history of the time taken to remediate (ART).
    - **Cons:** Requires strict governance to ensure tickets are not prematurely closed or deferred without executive **Risk Acceptance**.

6.6.2 Risk Scoring (e.g., Common Vulnerability Scoring System (CVSS))

- **Core Concept:** Applying a quantitative, standardized metric to assess the technical severity of a vulnerability, aiding in prioritization across different application teams.
- **Common Vulnerability Scoring System (CVSS):** The industry standard framework for measuring technical severity. It combines a set of metrics (Base, Temporal, and Environmental) to produce a numerical score (0.0 to 10.0), which maps to qualitative ratings (Low to Critical).
    - **Usage:** The CVSS score is the primary factor used to inform the **Break/Build Criteria** (Domain 2.3) and prioritize the fix against the **Business Risk** (Domain 2.8.4.1).
- **Implementation:** The CVSS score must be calculated for every security vulnerability found in testing. The **Environmental Score** is crucial in software security because it allows the team to adjust the severity based on the specific **Confidentiality, Integrity, and Availability (CIA)** loss to the *application's data* (Domain 3.3).
- **Pros/Cons:**
    - **Pros:** Provides an objective, standardized language for risk; essential for communication with automated tools.
    - **Cons:** Measures *technical* severity, not *business* impact; requires manual adjustment (Environmental Score) for accurate organizational context.

Domain 6.7 Secure Test Data

Definition

Establishing strict policies and technical mechanisms to ensure that sensitive data used in non-production environments (Dev, QA, Staging) adheres to privacy laws and organizational policy, preventing data leakage outside of the secure perimeter.

6.7.1 Generate Test Data

- **Core Concept:** Creating new, non-sensitive data sets that possess the required characteristics of real data without containing any actual PII.
- **Generation Focus:**
  - **Referential Integrity:** Ensuring that relationships between data elements (e.g., a customer ID links correctly to an order ID) are maintained, allowing the application logic to function correctly.
  - **Statistical Quality:** Ensuring the generated data mirrors the distribution and characteristics of production data (e.g., a proportional mix of user roles, transaction types) for realistic performance and logic testing.
  - **Production Representative:** The synthetic data must accurately represent the *volume* and *complexity* of production data to ensure the system meets **Non-Functional Requirements (NFRs)** for scalability and performance (Domain 6.1.3).
- **Implementation:** Using specialized data generation tools that create synthetic data based on database schemas.

6.7.2 Reuse of Production Data

- **Core Concept:** The high-risk process of using copies of real production data in non-production environments. This is often necessary for integration and complex functional testing but requires aggressive security controls.
- **Mitigation Techniques:**
  - **Obfuscation/Sanitization:** Modifying PII fields (e.g., changing names to random strings, altering dates of birth) to disrupt the direct link to a real person.
  - **Anonymization:** Irreversibly altering PII such that the data subject cannot be identified by any means (Domain 3.4.2.2).
  - **Tokenization:** Replacing PII with non-sensitive tokens while maintaining functional properties (Domain 5.1.11).
  - **Data Aggregation Mitigation:** Ensuring the data set is large enough that individual records cannot be identified by correlation or aggregation attacks.

- **Use Cases/Considerations:**
    - **Pros:** Production data provides the highest confidence in testing complex business logic.
    - **Cons:** If not rigorously secured, using production data in test environments constitutes a major **Confidentiality/Privacy** violation, potentially leading to regulatory fines (GDPR, HIPAA).

Domain 6.8 - Perform Verification and Validation Testing

Definition

Verification and Validation (V&V) is the final, formal quality assurance step in the testing phase. It ensures the application is built **correctly** (Verification) and that it meets the customer's security requirements and intended use (**Validation**).

Core Concept

V&V answers two separate, critical questions:

1. **Verification:** "Are we building the product, **right**?" (Checking technical compliance to specifications/design).
2. **Validation:** "Are we building the **right** product?" (Checking if the product meets the business need and user requirements).

6.8.1 Verification (Are we building the product, right?)

Definition

The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

- **Focus:** Checking that the code and design adhere to the **Security Architecture Document (SAD)** (Domain 4.5), **Coding Standards** (Domain 5.1), and **Non-Functional Requirements (NFRs)** (Domain 4.6).
- **Implementation:** Verification relies heavily on technical, internal activities:
    - **Unit Tests & Code Coverage:** Checking that the code for security functions (e.g., encryption, hashing) works as intended.
    - **Static Analysis (SAST):** Verifying adherence to coding standards and checking for basic implementation flaws (Domain 5.2).

    - **Peer/Code Review:** Ensuring the cryptographic calls and access control logic match the documented design (Domain 5.2).
    - **Security Configuration Review:** Checking that all necessary security headers and runtime settings are correct.

6.8.2 Validation (Are we building the right product?)

Definition

The process of evaluating software during or at the end of the development process to determine whether it satisfies specified user needs and security requirements.

- **Focus:** Checking the effectiveness against **Business Risk** (Domain 2.8) and **Functional Security Requirements (FSRs)** (Domain 3.1).
- **Implementation:** Validation involves external, user-centric, and security-centric checks:
    - **Acceptance Testing:** The final confirmation by the business owner or customer that the system meets all contractual and security requirements.
    - **Penetration Testing:** Validation that the **Residual Risk** (Domain 2.8) is acceptable by simulating a real attack.
    - **Misuse/Abuse Case Testing:** Validation that the system successfully prevents unauthorized usage scenarios (Domain 6.2).

6.8.3 Implementation Models for V&V

6.8.3.1 Independent/Internal Verification and Validation (IV&V)

- **Core Concept:** The process of performing V&V by a party that is separate from the development team. This separation ensures objectivity and reduces the likelihood that defects are overlooked due to familiarity.
- **Independent V&V:** Performed by an **external party** (e.g., a third-party security auditor or penetration testing firm). This provides the highest level of assurance, often required for high-risk systems.
- **Internal V&V:** Performed by a group within the organization but **separate from the development team** (e.g., the corporate Security Assurance team or a dedicated QA team). This maintains a degree of objectivity while keeping the process in-house.
- **Use Cases/Considerations:**
    - **Use When:** Mandatory for achieving **Authorization to Operate (ATO)** (Domain 2.9.4) for high-assurance or government systems.
    - **Pros:** Provides an objective assessment of security controls; essential for managing **Residual Risk**.

6.8.3.2 Acceptance Test

- **Core Concept:** The final validation phase where the client or product owner formally confirms that the delivered application meets the agreed-upon criteria and **Security Requirements**. A system cannot go live until security acceptance is granted.
- **Security Context:** The security acceptance test typically involves demonstrating that all **Critical** and **High-Risk** vulnerabilities found during testing have been mitigated, and the system performs securely under load (Non-Functional Requirements).
- **Use Cases/Considerations:**
  - **Pros:** The final **Control Gate** (Domain 2.3) that officially authorizes the system for production use, tying the testing results directly to business accountability.

Domain 7 – Secure Software Deployment, Operations, Maintenance – 11%

Domain 7.1 - Perform Operational Risk Analysis (11%)

Definition

Operational Risk Analysis is the systematic process of identifying, assessing, and mitigating security risks that arise specifically from **deploying and operating** the software in the live production environment. It focuses on risks related to the *environment*, *personnel*, *compliance*, and *integration* rather than just coding flaws.

7.1.1 Deployment Environment (e.g., staging, production, quality assurance (QA))

Core Concept

The security of an application is defined by the security of its least secure environment. Environments must be isolated, hardened, and strictly controlled to prevent unauthorized access or accidental exposure.

- **Isolation and Data Segregation:** Production, staging, and QA environments must be physically and logically **isolated** (using separate network segments or cloud accounts) to enforce **Least Common Mechanism** (Domain 1.2.8).
  - *Implementation:* Access controls must strictly enforce **Separation of Duties (SoD)** (Domain 1.2.2)—only DevOps/Ops teams can deploy to production; developers only push to staging/QA.
  - *Data Risk Mitigation:* **Never** use real **PII** (Personally Identifiable Information) in staging/QA environments. Data must be rigorously **anonymized or tokenized** (Domain 6.7) to prevent confidential information leakage during testing.
- **Production Environment Hardening:** The production environment must be hardened by disabling all unnecessary services, closing unused ports, and ensuring strict **Change Management** (Domain 2.9) is enforced for all configuration files.
  - *Example:* Network access to the production database should be restricted to the application server's IP address only, enforcing **Least Privilege** on network connectivity.

7.1.2 Personnel Training (e.g., administrators vs. users)

Core Concept

Security is a shared responsibility. Operational risk is minimized when all personnel understand their specific security roles and procedures relevant to the live environment.

- **Role-Based Focus:** Training must be tailored to the **level of access** and the tasks performed (Domain 2.2.2).
    - **Administrators/DevOps:** Need technical training on **Secure Configuration Management** (Domain 7.2), **Out-of-Band Management** (Domain 4.2.1.2), **Patch Management** (Domain 7.9), and forensic logging procedures.
    - **Standard Users:** Need training on recognizing phishing, secure usage of application features (e.g., MFA), and the correct process for reporting security incidents.
- **Incident Response Practice:** Operational staff must receive mandatory training and participate in **tabletop exercises** to practice executing the **Incident Response Plan** (Domain 2.9.2), ensuring a rapid and coordinated response during a breach.
    - *Example:* Training DevOps on the procedure to quickly isolate (quarantine) a compromised production server without disrupting the entire service (Containment).

7.1.3 Legal Compliance (e.g., adherence to guidelines, regulations, privacy laws, copyright, etc.)

Core Concept

The deployed application and its operational procedures must comply with all relevant legal, regulatory, and contractual mandates to avoid fines and legal penalties.

- **Data Residency and Jurisdiction:** Requires a final review to ensure the deployment configuration meets **Cross-Border Requirements** (Domain 3.4.4.2) for PII.
    - *Example:* Verifying that PII collected from EU users is stored and processed exclusively within EU-based data centers, as mandated by **GDPR**.
- **Copyright and Licensing:** The operational environment must adhere to all vendor and open-source licenses for the software components and infrastructure tools used.
    - *Risk:* Using software in a way that violates its license (e.g., deploying open-source components under a proprietary license without compliance) creates legal operational risk.
- **Operational Logging Compliance:** The logging procedures must be verified to ensure they meet **retention** (how long logs are kept) and **privacy** (PII masking in logs) requirements of relevant regulations (Domain 5.1.6).

7.1.4 System Integration

Core Concept

Analysing risks that arise specifically from the runtime communication between the application and other active systems (dependencies), often exposing **trust boundaries** that were not rigorously tested in isolation.

- **Emergent Risk:** New security risks often emerge when systems integrate for the first time in a live environment, such as performance bottlenecks or unforeseen denial-of-service conditions created by a dependency failure.
- **Trust Boundary Enforcement:** Focused **Integration Testing** (Domain 6.2) is mandatory for live interfaces.
    - *Mitigation:* Verifying that the application strictly validates all input from external dependencies (even internal microservices, adhering to **Zero Trust**) and that all data sharing is secured via **mTLS** or signed tokens (Domain 4.2.2).
    - *Example:* Analysing the risk that a failure in the external billing system could cause the application to crash or revert to an insecure, non-charged state (**Fail Secure** check).

Domain 7.2 Secure Configuration and Version Control

Definition

Establishing strict policies and tools to ensure that system configurations and software versions are managed securely, preventing unauthorized changes, ensuring integrity, and maintaining an auditable history of the system state.

7.2.1 Hardware and Baseline Configuration

- **Core Concept:** Securing the physical and virtual foundational components by enforcing a **hardened baseline configuration** for all application components (OS, web server, database) to prevent **Security Misconfiguration** (OWASP A05).
- **Implementation:** The baseline is typically defined using **Infrastructure as Code (IaC)** (e.g., Terraform, Ansible). This ensures secure defaults, such as disabling unnecessary services and ports, and configuring the application service to run as a non-root user (**Least Privilege**).
- **Use Cases and Pros/Cons: Pros:** Enforces **Consistency** across environments and prevents **configuration drift**; vital for continuous security. **Cons:** Initial development of the secure baseline templates is time-consuming.

7.2.2 Version Control/Patching

- **Core Concept:** Managing changes to software and configuration using a centralized **Version Control System (VCS)** and ensuring a timely and verified **Patch Management** process (Domain 7.9).

- **Implementation: All configuration files** and application code must be stored in the VCS. Patches are sourced from trusted vendors, tested in staging (**Integration Tests**), and their **integrity** is verified using cryptographic hashes before deployment to production. This mitigates the risk from **Vulnerable and Outdated Components** (OWASP A06).
- **Use Cases and Pros/Cons: Pros:** Maintains complete **Accountability** and traceability for every system state; reduces risk from outdated components. **Cons:** Requires strict **Change Management** (Domain 2.9) approval for all changes to prevent operational bottlenecks.

7.2.3 Documentation Practices

- **Core Concept:** Ensuring all changes to configuration and code are tracked and documented, linking them to an approved ticket and security review for **Accountability** and **Traceability** (Domain 3.7).
- **Implementation:** Every deployment artifact and configuration change must be linked to a formal **Release Note** (Domain 6.3) and recorded in the VCS. This creates an auditable history necessary for compliance and forensic analysis.

Domain 7.3 Release Software Securely

Definition

Implementing controls within the Continuous Integration/Continuous Delivery (CI/CD) pipeline to ensure that the final software package (artifact) is securely built, verified, and deployed without compromising the production environment.

7.3.1 Secure CI/CD Pipeline (e.g., DevSecOps)

- **Core Concept:** Integrating security tools and checks (e.g., **SAST**, **SCA**) directly into the automated pipeline (**Shift Left**). The pipeline environment itself must be treated as a critical asset and protected against attack.
- **Implementation Focus:**
  - **Least Privilege/SoD:** Using separate, dedicated **service accounts** for the build and deployment stages. The build account should not have permission to deploy, and the deployment account should not have permission to modify source code.
  - **Control Gates:** Security tools enforce **Break/Build Criteria** (Domain 2.3), automatically failing the deployment if a critical flaw is found (e.g., a high-severity CVE detected by SCA).
- **Use Cases and Pros/Cons: Pros:** Accelerates the **Average Remediation Time (ART)** (Domain 2.5); minimizes risk by automating human security checks. **Cons:** A vulnerability

in the build tools or service accounts can lead to a devastating **Software Supply Chain** attack.

7.3.2 Application Security Toolchain

- **Core Concept:** The automated sequence of security verification tools (SAST, DAST, SCA) used within the CI/CD pipeline to continuously verify the application's security posture.
- **Vetting and Configuration:** Ensuring the tools are configured correctly (e.g., not scanning non-production resources) and that the tools themselves are running the latest, secure versions. The outputs of this toolchain feed directly into the **Bug Tracking** system (Domain 6.6).
- **Implementation:** The toolchain should be configured for comprehensive coverage, running **SAST** on every code commit and **DAST** against the application once it is built and deployed to the staging environment.

7.3.3 Build Artifact Verification (e.g., code signing, hashes)

- **Core Concept:** Cryptographically verifying the final software package before it reaches the production server to ensure its **Integrity** and **Provenance** (Domain 8.3). This prevents a **Supply Chain** attack where a legitimate artifact is replaced with a malicious one.
- **Implementation:**
    - **Code Signing:** A trusted signing service applies a **digital signature** (**Code Signing**, Domain 5.6) to the artifact. The production server verifies this signature before execution, providing **non-repudiable proof** of the artifact's source.
    - **Hashes:** The system generates a cryptographic **hash** (e.g., SHA-256) of the artifact. This hash is stored in a secure ledger, allowing downstream systems to verify the artifact's **Integrity** at any point before execution.
- **Use Cases and Pros/Cons: Pros:** Provides non-repudiable proof of the artifact's source and integrity; critical defense against advanced supply chain compromise. **Cons:** Requires rigorous management of the private signing keys, which themselves become critical secrets.

Domain 7.4 - Store and Manage Security Data

Definition

Establishing rigorous policies and using specialized infrastructure to protect the most sensitive data required for application operation and security enforcement. This ensures the **confidentiality** and **integrity** of keys, secrets, and credentials throughout the operational lifecycle.

7.4.1 Secure Storage Architecture and Governance

Core Concept

Security data (credentials, keys, secrets) must be isolated from the application code and managed centrally using dedicated solutions like **Secrets Vaults** and **Hardware Security Modules (HSMs)**. This is a crucial implementation of the **Least Common Mechanism** principle (Domain 1.2.8).

- **Storage Location and Access:** Secrets must **never be hardcoded** or stored in version control systems (VCS). They must be retrieved dynamically at runtime from a secure, authenticated external service. Access to these stores must be strictly enforced using the principle of **Least Privilege** (Domain 1.2.1), restricted by network or **Trust Zones** (Domain 5.1.14).
- **Key Rotation:** The operational architecture must mandate and support **automatic, periodic replacement** of cryptographic keys and credentials. This process should support **zero-downtime rotation**, where new secrets are introduced without interrupting service, ensuring the continued **Availability** of the application.
    - *Pros:* Reduces the **blast radius** of a compromise, as the secrets are isolated; achieves compliance with regulatory key management standards.
    - *Cons:* **Complexity:** Introduces latency and requires the application to handle complex authentication and key retrieval logic. The secrets vault/HSM becomes a high-value, **single point of failure (SPOF)** that must be protected with maximum redundancy.

7.4.2 Specific Security Data Types and Management

7.4.2.1 Credentials

- **Core Security Requirement: Confidentiality** and **Nonrepudiation**. These are data used for **Authentication** (e.g., database user IDs and passwords for service accounts).
- **Management Technique:** Stored in **Secrets Vaults** (e.g., HashiCorp Vault). Access is logged, and credentials must be **automatically rotated** (e.g., every 90 days) without requiring developer intervention.

7.4.2.2 Secrets

- **Core Security Requirement: Confidentiality** and **Integrity**. This includes general sensitive information that must be hidden from developers and attackers (e.g., API keys for external services, third-party integration tokens).
- **Management Technique:** Secrets should be **dynamically provisioned** or retrieved only at the moment they are needed and kept in memory for the shortest possible time (**Just-in-Time access**). **Auditing** all access and retrieval attempts is mandatory (Domain 5.1.6).

7.4.2.3 Keys/Certificates

- **Core Security Requirement: Confidentiality, Integrity, and Availability.** These are cryptographic artifacts used for encryption, decryption, and identity verification (e.g., TLS/SSL certificates, signing keys, encryption master keys).
- **Management Technique (HSMs):** Stored within **Hardware Security Modules (HSMs)**. These are dedicated, tamper-resistant physical devices that securely store and perform cryptographic operations with keys. This provides the strongest **Root of Trust** (Domain 4.3.5) and protects keys from exfiltration.
- **Management Technique (CMS):** A **Certificate Management System** manages the entire lifecycle (issuance, revocation, renewal) of **X.509 certificates** (Domain 4.3.1) used for transport encryption and code signing.

7.4.2.4 Configurations

- **Core Security Requirement: Integrity** and **Availability.** Security-relevant configuration parameters that define the system's operational security posture (e.g., session timeout limits, logging levels, permitted cipher suites).
- **Management Technique:** Security configurations must be stored in **Version Control (VCS)** and managed via **Infrastructure as Code (IaC)** (Domain 7.2). All changes must follow a strict **Change Management Process** (Domain 2.9.1) with mandatory security review.

Domain 7.5 - Ensure Secure Installation

Definition: The process of deploying the software into its final operational environment using controls and procedures that guarantee the integrity of the installed code and enforce the required security posture from the first moment of execution.

7.5.1 Secure Boot and Platform Integrity

Core Concept

Ensuring that the underlying hardware platform is in a trusted, unmodified state before executing the application. This is a foundational control for **Integrity** and **Provenance** (Domain 8.3).

- **Secure Boot:** A security standard implemented in a computer's firmware (UEFI) that verifies the digital signature of the operating system's loader and kernel before allowing them to execute.
    - **Key Generation, Access, and Management:** The entire Secure Boot process relies on **cryptographic keys** (PKI). These keys must be securely generated, their access must be restricted to authorized parties, and their management (e.g., storage in the **TPM** or **Secure Element**, Domain 4.1.10) must follow strict lifecycle policies (Domain 7.4.2.3).
    - **Goal:** Prevents the injection of unauthorized or malicious code (e.g., rootkits or unauthorized bootloaders) early in the boot process.
- **Implementation:** The application may verify the state of the **Trusted Platform Module (TPM)** (Domain 4.3.5) on startup to ensure its environment has not been tampered with.

7.5.2 Principle of Least Privilege (PoLP)

Core Concept

The installation must ensure that the application and its dedicated service accounts are granted only the absolute minimum access rights and permissions necessary to perform their defined functions, and no more.

- **Service Account Provisioning:** The installation scripts must create dedicated **Service Accounts** (Domain 3.5.2) with granular permissions. *Example:* The application's web service should run as a non-root user and only have read access to configuration files and write access to its designated log directory.
- **Access Restriction:** The installation process explicitly restricts system resources (e.g., file system access, network socket binding) that are not strictly necessary for the application's operation.
- **Principle Applied:** Enforcing **PoLP** is a critical **Preventive Control** that limits the **blast radius** if the application is compromised.

7.5.3 Environment Hardening

Core Concept

Applying rigorous security configurations to the operational environment (OS, network, and hosting platform) to reduce the overall attack surface and mitigate common configuration risks.

- **Configuration Hardening:** Disabling unnecessary services, closing unused ports, removing default or sample accounts, and configuring secure logging parameters. This ensures the environment starts from a **Secure Baseline Configuration** (Domain 7.2.2).

- **Secure Patch/Updates:** The installation procedure must include verifying the current patch level of the OS and runtime environment, applying any missing critical updates, and establishing an automated **Patch Management** process for the future (Domain 7.9). Updates themselves must be verified using **Build Artifact Verification** (Domain 7.3.3) before application.
- **Firewall:** Implementing network segmentation rules (e.g., **Least Common Mechanism**, Domain 1.2.8) and host-based firewalls that restrict traffic to only necessary ports and protocols required for the application to function. *Example:* Restricting database connections to only the application server's internal network IP.

7.5.4 Secure Provisioning

Core Concept

Automating the secure setup of the application and its environment using code-based, repeatable processes, minimizing human error and ensuring rapid, auditable deployment.

- **Infrastructure as Code (IaC):** Using tools (e.g., Terraform, CloudFormation) to define, provision, and configure the underlying environment securely. IaC enforces the **Security Policy** from code, ensuring consistency and preventing manual **Configuration Drift** (Domain 7.2).
- **Credentials Provisioning:** The installation process must dynamically retrieve **Credentials** and **Secrets** from a central **Secrets Vault** (Domain 7.4), never relying on manual entry or hardcoded defaults.
- **Configuration and Licensing:** Provisioning the application with all necessary security configurations and ensuring all required **licensing** is in place before the system goes live, adhering to contractual requirements (Domain 3.8.3).

7.5.5 Security Policy Implementation

Core Concept

Ensuring that the security requirements defined during the requirements and design phases are enforced via technical controls in the live environment during installation.

- **Enforcement:** The installation process must confirm that controls derived from the **Security Architecture Document (SAD)** (Domain 4.5) are active. *Example:* Verifying that the correct **TLS certificates** are installed and that the application server is configured to reject weak cipher suites.
- **Final V&V:** The installation ends with a formal **Verification and Validation (V&V)** test (Domain 6.8) to confirm that the security policy is correctly applied before granting **Authorization to Operate (ATO)** (Domain 2.9.4).

Domain 7.6 - Obtain Security Approval to Operate

Definition: The final, mandatory governance process where a formal decision-maker reviews the application's final security posture, determines that the **Residual Risk** is acceptable, and grants official permission for the system to enter production. This process is commonly known as **Assessment and Authorization (A&A)**.

7.6.1 Risk Acceptance

Core Concept

The formal acknowledgment by the designated authority that the system, despite having known, unmitigated vulnerabilities (the **Residual Risk**), is secure enough to operate based on organizational risk tolerance and business needs.

- **Residual Risk:** This is the level of risk remaining **after** all security controls have been implemented, tested, and validated (Domain 2.8.1.2). The goal of the entire **Verification and Validation (V&V)** process (Domain 6.8) is to measure and document this exact risk.
- **Risk Documentation:** The risk acceptance process is driven by the **Risk Register** (Domain 2.8), which lists:
    - All identified vulnerabilities (from Penetration Tests, DAST, etc.).
    - The **Criticality Level** or CVSS score for each vulnerability (Domain 6.6.2).
    - The mitigation strategy chosen (e.g., Fix, Transfer, or **Accept**).
    - The justification for accepting the risk (e.g., cost of fix exceeds ALE, fix is technically infeasible).

Implementation and Examples

- **Security Review Board (SRB) Vetting:** The security team presents the final **Residual Risk** report to the SRB. The report might show 1 Critical, 5 High, and 20 Medium vulnerabilities were fixed, but 2 Low-risk vulnerabilities have been explicitly deferred (accepted).
- **Risk Ownership:** The **Data Owner** or **Business Owner** (Domain 3.3.1.1) must confirm they understand the business consequences of the accepted risks. For instance, accepting the risk of a minor data integrity flaw because the cost of fixing it ($500,000) is far higher than the calculated Annualized Loss Expectancy (ALE, Domain 2.8).

7.6.2 Sign-Off at Appropriate Level

Core Concept

The Authorization decision must be made by an individual with the authority and organizational accountability to officially take on the **business risk** represented by the application.

- **Authorization to Operate (ATO):** This is the formal document or status granted by the senior official. Once the ATO is granted, the system can go live.
- **The Authorizing Official (AO):** This senior individual (e.g., the Chief Information Security Officer (CISO), a Vice President, or a designated business owner) is the only one who can grant the ATO. Their seniority level corresponds directly to the maximum **Business Risk** the system is designed to handle.
    - *Example:* A system handling national security data requires a very senior government official as the AO, whereas an internal HR system may only require the department head.
- **Principle Applied:** Enforces **Accountability** (Domain 1.1.6) and **Segregation of Duties (SoD)** (Domain 1.2.2) by ensuring the person who develops and assesses the system (development/security teams) does *not* make the final authorization decision.

Use Cases and Pros/Cons

Aspect	**Security Approval / ATO**
Use Case	Mandatory final **Control Gate** (Domain 2.3) for any system deployment, especially those governed by mandates like NIST RMF or ISO 27001.
Advantage	**Formalizes Accountability:** Ensures the CISO or business executive formally owns the **Residual Risk** before the system goes live.
Disadvantage	The ATO process can be seen as a time-consuming bureaucratic hurdle, potentially delaying critical business launches if security analysis is not performed early (Shift Left).

Domain 7.7 - Perform Information Security Continuous Monitoring

Definition: The systematic, ongoing process of collecting, analyzing, and reporting security-relevant data about the application and its environment to detect threats, maintain compliance, and verify the continued effectiveness of security controls. Continuous monitoring is the **Detective** and **Corrective** phase of the operational lifecycle.

7.7.1 Observable Data and Telemetry

Core Concept

Designing the application to produce comprehensive data streams that provide visibility into its runtime behavior, enabling security analysts to identify anomalies and breaches.

- **Logs and Events (Security Logging):** The application must be designed to generate specific records of all **security-relevant events** (e.g., failed logins, authorization denials, configuration changes). These logs must adhere to **secure logging practices** (Domain 5.1.6), ensuring they are protected from tampering and do not expose PII.
- **Telemetry and Trace Data:** Automated data collected from application components regarding their health, performance, and internal execution paths. **Trace Data** allows security teams to follow a request through a complex **microservices architecture** (Domain 4.1.4) to understand exactly where a transaction failed or was compromised.
- **Metrics:** Real-time data streams that quantify system activity (e.g., number of active sessions, CPU load, network I/O). Security metrics can include the rate of login failures or the volume of traffic crossing a critical **Trust Boundary** (Domain 4.2).
- **Implementation:** Using open standards (e.g., OpenTelemetry, common log formats) for collection and transfer to ensure seamless **Integration Analysis** (Domain 7.7.5).

7.7.2 Threat Intelligence

Core Concept

The proactive integration of external knowledge about current and emerging threats (actors, vulnerabilities, and attack methods) into the continuous monitoring platform to improve detection capabilities.

- **Usage in Monitoring:** Threat intelligence feeds (e.g., malicious IP addresses, known malware hashes) are ingested by the **SIEM** system and automatically correlated with the application's logs and events.
- **Action:** If an application log shows an attempted login from an IP address identified in a threat intelligence feed, the system can automatically flag or block the traffic, accelerating the **Intrusion Detection/Response** (Domain 7.7.3).

7.7.3 Intrusion Detection/Response (IDR)

Core Concept

The mechanisms and policies used to identify suspicious activities or policy violations (**Detection**) and the immediate steps taken to stop or minimize the damage (**Response**).

- **Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS):** Network or host-based systems that monitor traffic and system activity for patterns indicative of an attack (e.g., scanning, exploit attempts). IPS takes automated action (e.g., dropping packets, blocking an IP address) to stop the threat in real-time.
- **Runtime Application Self-Protection (RASP):** A security control deployed *inside* the application runtime environment (Domain 7.11). RASP continuously monitors application behavior and can block malicious input (e.g., SQL injection) and attempted attacks from within the application process itself, providing immediate **Corrective Control**.
- **Response:** If an intrusion is confirmed, the IRP (Incident Response Plan, Domain 2.9) is immediately executed, focusing first on containment and eradication.

7.7.4 Regulation and Privacy Changes

Core Concept

Monitoring the external regulatory environment to ensure that the deployed software and its operational procedures remain legally compliant.

- **Privacy Changes:** Tracking updates to laws like **GDPR, CCPA, or HIPAA** (Domain 3.2). A change in a data residency law might require an immediate, architectural change to how and where PII is stored.
- **Policy Verification:** Continuous monitoring is used to verify that internal security policies (e.g., password complexity, use of approved cipher suites) are being adhered to by the running system and have not suffered **configuration drift** (Domain 7.2.2).

7.7.5 Integration Analysis (e.g., Security Information and Event Management - SIEM)

Core Concept

The central aggregation and analysis point for all collected security data. The SIEM is critical for correlating disparate events and transforming raw logs into actionable intelligence.

- **Security Information and Event Management (SIEM):** A platform that collects logs and event data from all sources (firewalls, applications, servers, databases, IDPS) and uses rules and analytics to:
  - **Analyze:** Detect patterns that indicate a complex attack spanning multiple systems (correlation).
  - **Alert:** Generate high-priority notifications for the security operations center (SOC) to investigate.

- **Application Integration:** The application's log interface (Domain 4.2.1.3) must be configured to format and transmit logs to the SIEM in a consistent, standardized, and secure manner (e.g., using an encrypted, reliable logging protocol).
- **Outcome:** The SIEM transforms the raw data into evidence for **Accountability** and **forensic analysis**, directly supporting the **Incident Response Plan**.

Domain 7.8 - Execute the Incident Response Plan

Definition: The systematic execution of the pre-defined **Incident Response Plan (IRP)** (Domain 2.9.2) upon detection of a confirmed security breach. The process is designed to minimize damage, rapidly restore system security, and learn from the failure.

7.8.1 Incident Triage

Core Concept

The immediate process of rapidly assessing a potential security event to determine its validity, severity, and priority. Triage determines if an alert is a genuine **security incident** (requiring full IRP execution) or a false positive/minor event.

- **Flow/Steps:** Triage typically involves **Detection -> Verification -> Prioritization**.
    - **Detection:** An alert is received from a **SIEM** (Domain 7.7.5), IDS/IPS, or monitoring system.
    - **Verification:** Security analysts confirm the alert is not a false positive (e.g., confirming malicious activity is actually occurring).
    - **Prioritization:** The incident is scored based on the **Criticality Level** (CVSS, Domain 6.6.2) and the **Business Risk** (Domain 2.8) of the affected system and data. High-priority incidents (e.g., active data exfiltration) trigger immediate containment.
- **Software Application Context:** Triage determines if 5,000 failed login attempts are a simple automated bot attack (low priority, block IP) or a targeted credential stuffing attack on the admin panel (high priority, trigger IRP).
- **Goal:** To decide quickly on the necessary **Containment** actions.

7.8.2 Forensics

Definition

The systematic collection, preservation, analysis, and interpretation of digital evidence related to a security incident, performed to reconstruct the attack, identify the attacker, and understand the scope of the compromise.

- **Core Concept:** Maintaining the **Chain of Custody** for all evidence is paramount to ensure its legal admissibility (Domain 2.8.4.2). The integrity of the evidence must be provable (**Integrity**, Domain 1.1.2).
- **Evidence Collection:** Involves collecting volatile data (e.g., running processes, system memory, network connections) before non-volatile data (e.g., hard drive images, log files).
    - **Application Logs:** Forensics relies heavily on the quality and integrity of the application's **secure logs and audit trails** (Domain 5.1.6). Logs must be time-synchronized and protected from modification.
    - **System Images:** Creating bit-for-bit copies (images) of affected servers and systems for detailed, offline analysis.
- **Analysis:** Identifying the attacker's **Tactics, Techniques, and Procedures (TTPs)**, determining the initial exploit vector, and verifying the scope of **Information Disclosure** (Confidentiality loss, Domain 1.1.1).

7.8.3 Remediation

Definition

The process of eliminating the cause of the incident and restoring the system to a secure, pre-incident state. This is often an iterative process tied closely to the forensics team's findings.

- **Containment:** The immediate action taken to limit the damage and stop the attack from spreading (e.g., isolating the compromised server, revoking compromised API keys, disabling network access). This is a temporary measure.
- **Eradication:** The process of removing the root cause of the incident (e.g., patching the vulnerable code, removing the backdoor/malware, forcing password resets for compromised accounts).
- **Validation:** After remediation, the system must undergo a full **Verification and Validation (V&V)** test (Domain 6.8) to confirm that the vulnerability is closed and no new flaws were introduced.
- **Software Application Context:** If the root cause was an SQL Injection flaw, the code must be patched to use **Parameterized Queries** (Domain 5.1.2) and redeployed through the secure **CI/CD pipeline** (Domain 7.3).

7.8.4 Root Cause Analysis (RCA)

Definition

The final, crucial step after the incident is contained and remediated. RCA identifies the fundamental failure—whether procedural, human, or technical—that allowed the incident to occur.

- **Core Concept:** RCA moves beyond the immediate fix (the symptom) to identify the true underlying issue (the cause). It is the source of **lessons learned** that drive continuous security improvement (Feedback Loops, Domain 2.7).
- **Process:** Often conducted in a **post-mortem review** involving all teams (Dev, Sec, Ops, Management). It answers questions like:
  - *Why* did the SAST tool fail to find the flaw? (Tool/Process failure)
  - *Why* did the developer use an insecure function? (Training failure)
  - *Why* was the service account running with excessive privileges? (**Least Privilege** failure)
- **Outcome:** The RCA results in actionable recommendations that feed directly back into the **Requirements** (Domain 3) and **Secure Coding Standards** (Domain 5.1), preventing recurrence and improving overall organizational **Resiliency**.

Domain 7.9 Perform Patch Management

Definition

The systematic process of identifying, acquiring, testing, and deploying updates (patches) to software, operating systems, and firmware to correct security flaws, bugs, or improve performance.

Core Concept

Patch management is a critical **Corrective Control** that reduces the **attack surface** by mitigating known vulnerabilities discovered post-release. Timeliness and integrity are key to reducing the **Average Remediation Time (ART)** (Domain 2.5).

Implementation and Examples

- **Secure Release:** Patches must be sourced directly from the original vendor or supplier and verified using **cryptographic hashes or digital signatures** (Domain 7.3.3) before deployment. *Goal:* Prevent an attacker from injecting malicious code into the patch update itself (Supply Chain Attack).
- **Testing:** Patches must be thoroughly tested in a dedicated staging or QA environment (Domain 6.1.5) to ensure they do not introduce new functionality bugs or cause security regressions (Domain 6.2.4.3). This is particularly important for critical security controls.
- **Automation:** Utilizing automated tools for patch deployment (e.g., configuration management tools) to ensure consistency and speed across the entire fleet of servers.

- **Software Application Context:** The operations team must rapidly deploy a vendor patch to the application server's operating system to fix a recently announced **Critical** vulnerability (CVSS 9.8).

Domain 7.10 Perform Vulnerability Management

Definition

The cyclical process of identifying, classifying, prioritizing, remediating, and reporting on vulnerabilities in the software and its operational environment.

Core Concept

Vulnerability management is the engine of continuous security improvement, ensuring all identified flaws are systematically tracked until remediation or formal **Risk Acceptance** (Domain 7.6).

Implementation and Examples

- **Tracking:** All vulnerabilities (found via SAST, DAST, Pen Test, or public advisories) must be logged in a **Bug Tracking** system (Domain 6.6.1) and monitored until closure.
- **Triaging:** The process of rapidly validating and categorizing a vulnerability's severity and impact upon discovery. This involves calculating the **CVSS** score (Domain 6.6.2) and linking it to **Business Risk** (Domain 2.8.4.1) to determine the urgency of the fix.
- **Common Vulnerabilities and Exposures (CVE):** A standardized dictionary that assigns a unique identifier to publicly known information security vulnerabilities. *Usage:* The vulnerability management team uses CVEs to check if the application's third-party components (dependencies) are affected by publicly known flaws (Software Supply Chain Risk).
- **Software Application Context:** The security team receives an alert from an **SCA tool** (Domain 5.5.2) about a new CVE affecting a core logging library. The team triages the finding, calculates its severity, and assigns the remediation ticket to the development team for mitigation.

Domain 7.11 Incorporate Runtime Protection

Definition

Implementing security controls that actively monitor and protect the application during its execution, providing real-time defence against attacks that bypass perimeter controls.

Core Concept

Runtime protection provides dynamic **Detective** and **Preventive** controls by observing the application's behaviour in the high-risk "Data in Use" state (Domain 3.3.3).

Implementation and Examples

- **Runtime Application Self-Protection (RASP):** A security control deployed **inside the application runtime** (e.g., JVM or CLR). RASP continuously monitors application behaviour (e.g., function calls, data flow) and can block malicious input and attempted attacks from within the application process itself. *Example:* RASP detects an attempt at SQL Injection and blocks the request *before* it reaches the database, regardless of whether perimeter firewalls detected it.
- **Web Application Firewall (WAF):** A specialized type of reverse proxy deployed **in front of** the application. WAFs inspect HTTP/S traffic for attack patterns (e.g., XSS, SQLi, parameter tampering) and block malicious requests before they reach the web server.
- **Address Space Layout Randomization (ASLR):** An operating system mechanism that randomly arranges the positions of key data areas (executable code, libraries, stack, heap) in memory. *Purpose:* Makes memory corruption attacks (e.g., **buffer overflows** and **RCE**) much harder to reliably execute by preventing an attacker from guessing the necessary memory addresses.
- **Dynamic Execution Prevention (DEP):** A processor feature (often implemented via the **NX/XD bit**, Domain 5.1.15) that marks memory regions as non-executable. *Purpose:* Prevents attackers from injecting and running malicious code from data regions of memory (a classic defence against buffer overflows).

Use Cases and Pros/Cons

Aspect	**Runtime Protection**
Use Case	Essential for public-facing web applications; critical for protecting systems that process sensitive data.
Pros	Provides real-time, zero-day protection (especially RASP); acts as a **virtual patch** when code updates are not immediately possible.
Cons	WAFs can be bypassed; RASP can introduce performance overhead if not carefully tuned.

Domain 7.12 Support Continuity of Operations

Definition

Establishing the processes, documentation, and technical infrastructure necessary to ensure the application and business functions can withstand, rapidly recover from, and adapt to major disruptions (e.g., cyberattacks, natural disasters).

7.12.1 Backup, Archiving, and Retention

- **Core Concept:** Ensuring data **Integrity** and long-term **Availability** by securing data copies and defining their lifecycle.
  - **Backup:** A copy of data used for short-term restoration (e.g., recovering a corrupted file). Must be regularly verified for integrity and protected from malware (e.g., using **immutable storage**).
  - **Archiving:** Moving historical data off primary systems to secure, long-term storage to meet **Data Retention** policy requirements (Domain 3.4.4.1). Archives must be encrypted and isolated.
  - **Retention:** The policy defining how long data must be kept (minimum period for legal compliance) and when it must be securely **destroyed** (maximum period for privacy compliance, Domain 2.6.2.1).

7.12.2 Disaster Recovery Plan (DRP) and Business Continuity Plan (BCP)

- **Business Continuity Plan (BCP):** The high-level, business-focused plan detailing how critical **business functions** will continue during and after a major disruption. *Focus:* People, processes, and essential operations.
- **Disaster Recovery Plan (DRP):** The detailed, technical plan for recovering **IT infrastructure and applications** following a disaster. The DRP must align with the BCP and include measurable targets:
  - **Recovery Time Objective (RTO):** The maximum tolerable duration of time in which a system or application can be down after a failure.
  - **Recovery Point Objective (RPO):** The maximum tolerable period in which data might be lost from an IT service due to a major incident.

7.12.3 Resiliency

- **Core Concept:** The ability of the application architecture to anticipate, absorb, and adapt to disturbances (failures or attacks) while continuing to operate at an acceptable level.

- **Operational Redundancy:** Deploying redundant components (e.g., active-active clusters, geo-distributed load balancing) to eliminate **Single Points of Failure (SPOFs)** and maintain service availability (Domain 4.6.1.3).
- **Erasure Code:** An advanced data integrity technique that fragments data into pieces and encodes it with redundant parity data, allowing the original data to be reconstructed even if some fragments are lost. *Focus:* High-integrity storage resilience.
- **Survivability:** The ability of a system to continue to perform essential functions in the presence of severe attacks or faults, particularly a **Denial-of-Service (DoS)** attack. *Implementation:* Strong rate limiting, capacity planning, and automated failover.

Domain 7.13 Integrate Service Level Objectives and Service-Level Agreements (SLA)

Definition

Formalizing performance, security, and maintenance expectations into measurable agreements, ensuring that security-related operational duties are met by both internal teams and external vendors.

Core Concept

An **SLA** is a legally binding contract (Domain 3.8.3) between a service provider and a customer (internal or external). **SLOs (Service Level Objectives)** are the internal goals set to meet or exceed the customer-facing SLA thresholds.

Implementation and Examples

- **Performance and Availability:** The SLA must define the required **uptime percentage** (e.g., 99.99%) and acceptable response times. This places a security requirement on development and operations to prevent performance-impacting flaws.
- **Maintenance:** Defines the required **patching frequency** (Domain 7.9), the maximum allowable downtime for scheduled maintenance, and the required rollback procedures. *Security Context:* The SLA may require critical security patches to be deployed within 72 hours of release.
- **Qualified Personnel:** Ensures that the vendor or internal operations team responsible for maintenance and incident response possesses the necessary security certifications and expertise. *Context:* An SLA may require that all system administrators managing the application have the **CSSLP** or **CISSP** certification.
- **Incident Response Integration:** The SLA/SLO must include security-specific metrics, such as:
    - Maximum time to notify the customer of a breach.

    - Maximum time to isolate a compromised system (Containment).

Use Cases and Pros/Cons

Aspect	**SLOs and SLAs**
Use Case	Mandatory for managing vendor services (SaaS, Cloud) and for governing internal service delivery.
Pros	**Formalizes Accountability** (Domain 1.1.6); provides measurable, data-driven targets for security and operations teams.
Cons	Failure to meet SLA thresholds can result in financial penalties for the service provider.

Domain 8 – Secure Software Supply Chain – 10%

Domain 8.1 - Implement Software Supply Chain Risk Management (10%)

Definition

Software Supply Chain Risk Management (SCRM) is the process of identifying, assessing, and mitigating the security risks associated with every external component (software, services, hardware) and every phase (design, development, distribution) that contributes to the final software product. This is crucial as most modern applications are built using over 80% third-party code.

8.1.1 Governance and Standards

Core Concept

Establishing a formal governance framework to manage third-party risk, ensuring continuity and compliance.

- **International Organization for Standardization (ISO):** Using standards like **ISO 27036** (Security techniques - Supplier relationships security) to guide the security management of supplier agreements and component acquisition. This provides a globally recognized, auditable process.
- **National Institute of Standards and Technology (NIST):** Utilizing guidance from **NIST SP 800-161** (Supply Chain Risk Management Practices) to establish a comprehensive framework for managing the risk of compromise at any point in the software's delivery chain.

8.1.2 Identification and Selection of the Components

Core Concept

Proactively vetting components before they are integrated, ensuring they meet minimum security and compliance standards.

- **Vetting Source and Trustworthiness:** Components must be sourced from reputable, actively maintained repositories or vendors (Domain 3.8). Reviewing the history and community support of open-source libraries is a key step.
- **Licensing and Legal Compliance:** The selection process must involve legal review to ensure the component's license (e.g., GPL, MIT, Apache) is compatible with the proprietary application's license (Domain 3.8.3). Violating license terms is a significant legal risk.

- **Functionality Scoping:** Strictly limiting component usage to only the functionality required (**Least Privilege** for code). Choosing smaller, single-purpose components over large, monolithic ones reduces the attack surface.
- **Software Application Context:** Before selecting a new JSON parsing library, the team checks its maintenance history, confirms it has no known severe **CVEs** (Common Vulnerabilities and Exposures), and verifies its open-source license is acceptable.

8.1.3 Risk Assessment of the Components

Definition

Formally evaluating the level of security risk associated with integrating a component into the application.

- **Vulnerability Assessment:** Running **Software Composition Analysis (SCA)** (Domain 5.5.2) against the component to identify any known **CVEs** that exist in the codebase.
- **Security Testing:** Performing dynamic testing (DAST) or even static analysis (SAST) on the component's core functionality (if possible) to find flaws not yet publicly disclosed.
- **Decision Strategy (Risk Treatment):** Based on the assessment, a decision must be formally documented (Domain 2.8):
  - **Mitigate:** The most common approach. Updating the component to a secure version, applying a patch, or wrapping the component with defensive code (virtual patching) if a fix isn't available.
  - **Accept:** Formal management sign-off to accept the **Residual Risk** posed by a vulnerability (usually for low-severity or non-exploitable flaws, or when the cost of mitigation is too high). This acceptance must be regularly reviewed (Domain 7.6).
  - **Avoid:** Choosing a different, more secure component or building the required functionality internally.

8.1.4 Maintaining Third-Party Components List (Software Bill of Materials)

Definition

Creating and maintaining a complete, machine-readable inventory of all third-party and open-source components used in the application.

- **Software Bill of Materials (SBOM):** A complete, formal list that tracks every component, version number, license, and vendor used in the final software package. *Purpose:* The SBOM is essential for rapid response during a major supply chain event.
- **Implementation:** The SBOM is typically generated automatically during the **build process** (Domain 7.3) using SCA tools. It must be stored securely and made available for auditing and rapid vulnerability analysis.

- **Use Case (Vulnerability Response):** If a new zero-day vulnerability (e.g., Log4Shell) is announced, the security team can immediately query the SBOM to identify every single application that uses the affected component, rather than manually searching repositories.

8.1.5 Monitoring for Changes and Vulnerabilities

Definition

Implementing continuous **detective controls** to ensure the security status of components is verified throughout the application's entire operational lifecycle, not just at initial integration.

- **Continuous Scanning:** SCA tools must run continuously (or nightly/weekly) against the component inventory to check for newly published CVEs. This process must be integrated into the **Continuous Integration (CI)** pipeline (Domain 7.3.1).
- **Component Change Detection:** Monitoring for changes in the component itself (e.g., a vendor changing their license, a contributor being flagged as malicious, or a component being retired).
- **Risk Metrics:** Monitoring metrics like the **Average Remediation Time (ART)** (Domain 2.5) for supply chain vulnerabilities to ensure fast response when new, critical CVEs are announced.

Domain 8.2 - Analyse Security of Third-Party Software

Definition

The formal process of performing **due diligence** on external software components and services **before** acquisition or integration. This process verifies the vendor's security capabilities and the inherent safety of their product.

8.2.1 Attestation and Assurance

Core Concept

Collecting and reviewing independent, third-party proof (attestations) that the vendor's organization and product security controls are effective. Reliance on vendor-supplied information must be minimized.

- **Certifications:** Official validation that the vendor's Information Security Management System (ISMS) adheres to recognized international standards. *Example:* Reviewing the vendor's **ISO 27001 certification** to confirm they have implemented a formal program for vulnerability management and secure development (Domain 2.2.3).

- **Assessment Reports (e.g., cloud controls matrix):** Detailed audit reports prepared by independent third parties. These provide concrete evidence of control effectiveness.
    - **SOC 2 Type 2 Reports:** Provides assurance over a vendor's internal controls (security, availability, processing integrity, confidentiality, and privacy) over a period of time. Essential for cloud service providers.
    - **Cloud Controls Matrix (CCM):** A specific framework developed by the **Cloud Security Alliance (CSA)** that lists fundamental security principles and controls for cloud vendors, used to guide controls assessment.
- **Security Context:** Attestation helps manage **Residual Risk** (Domain 2.8) because it provides independent proof that the vendor has implemented required security controls.

8.2.2 Origin and Support Vetting

Core Concept

Analysing the component's source, maintenance history, and the vendor's ability to provide timely security support, which directly impacts the application's **Availability** and **Vulnerability Management** (Domain 7.10).

- **Origin (Provenance):** Identifying the source of the software (vendor, open-source project, specific repository). *Goal:* To ensure the software has not been altered or compromised before reaching the organization (Supply Chain Integrity).
- **Support Structure:** Analyzing the vendor's capacity and process for delivering security support. This is critical for assessing risk before the component is integrated.
    - **Community vs. Commercial Support:** Open-source projects with only community support may have unpredictable patch timelines, increasing **risk exposure** when a critical vulnerability is found. Commercial support offers contractual guarantees (SLAs).
    - **Patching Commitment:** Reviewing the vendor's policy on vulnerability disclosure and commitment to fixing severe flaws (e.g., maximum time to patch critical CVEs).
- **Security Track Record:** Reviewing the vendor's historical performance on security disclosures, past breaches, and their response time to reported vulnerabilities. A poor track record indicates high operational risk.

8.2.3 Formal Requirements and Agreements

Core Concept

The security findings and requirements must be formalized into legally binding documents before a purchase or integration is finalized (Domain 3.8.3).

- **Audit of Security Policy Compliance:** Requiring the vendor to submit documentation proving their adherence to secure software development practices (e.g., using **SAST** and **Threat Modeling**) during their own development process.
- **Vulnerability/Incident Notification:** The contract (SLA/MSA) must explicitly require the vendor to **immediately notify** the customer in the event of a security incident or the discovery of a vulnerability in their code. The contract defines the mandatory **Incident Response** cooperation.
- **Scope of Testing (Shared Responsibility Model):** Clearly defining who is responsible for securing what, especially in cloud services (Domain 4.1.8). *Example:* The agreement clarifies that the customer is responsible for **application-level patching** (IaaS), while the vendor is responsible for **network security**.
- **Log Integration into SIEM:** Requiring the vendor's service to provide a secure, standardized **log interface** that allows the customer to ingest security events into their own **SIEM** system (Domain 7.7.5). This is vital for **continuous monitoring** and cross-application correlation.

8.2.4 Vendor Security Requirements in the Acquisition Process

- **Right to Audit:** Including a clause in the contract (Master Service Agreement) that grants the organization the right to perform its own security audits or request third-party penetration testing against the vendor's environment or code. This acts as a strong **deterrent** and ensures independent verification.

Domain 8.3 - Verify Pedigree and Provenance

Definition

The process of establishing and verifying the **history, integrity, and source** of a software component or artifact from its point of origin through every stage of the development and delivery lifecycle. This ensures that the final code executed has not been tampered with and comes from a trustworthy source.

8.3.1 Secure Transfer and Integrity

Core Concept

Ensuring that software components are protected against unauthorized modification or substitution while they are being moved between environments, teams, or vendors. This maintains **Integrity** and **Authenticity**.

- **Secure Transfer (Chain of Custody):** Establishing and documenting the complete, uninterrupted history of the artifact, tracking every entity (person, tool, or system) that has handled it.

    - *Implementation:* **Chain of Custody** requires formal procedures for logging when a component moves from one control zone to another (e.g., from the build server to the deployment repository).

- **Authenticity and Integrity:** Using cryptographic mechanisms to prove the component's origin and verify it hasn't been altered.
    - **Cryptographically-Hashed, Digitally-Signed Components:** The final build artifact must be signed by a trusted, secure key (Domain 7.3.3). This **digital signature** (Domain 1.1.2) provides **non-repudiable proof** (Domain 1.1.7) that the component originated from the trusted build system and its content has not changed.
    - *Verification:* Downstream systems (e.g., the production server) must verify the signature's validity before the component is executed.
- **Software Application Context:** A deployment pipeline verifies the **hash** of a new container image pulled from the internal registry and confirms the image was signed by the official corporate signing key before pushing it to the Kubernetes cluster.

8.3.2 Secure Environments and Isolation

Core Concept

Protecting the internal infrastructures where software is stored, built, and tested, as these are critical points of compromise in a supply chain attack.

- **Code Repository Security:** The source code repository (e.g., Git) is the highest-value asset, as its compromise directly enables attackers to inject backdoors.
    - *Implementation:* Requires strict **Access Control** (MFA, RBAC, Domain 3.5), detailed **Auditing** of all access and modification events, and isolation (e.g., separating internal Git from public networks).
- **Build Environment Security:** The infrastructure (servers, agents, compilers) used to compile the final executable. Compromise here leads to a full **Supply Chain Attack** (e.g., SolarWinds).
    - *Implementation:* Utilizing ephemeral, **isolated build agents** (e.g., containers or VMs) that are destroyed after every build (**Least Common Mechanism**, Domain 1.2.8) and enforcing **Secure Configuration Management** (Domain 7.2.2) on the build process tools.
- **System Sharing/Interconnections:** Securely managing the data and network interfaces between different internal systems (e.g., build server communicating with the dependency registry).
    - *Implementation:* All communication must use strong, authenticated protocols (e.g., **mTLS**) and adhere to **Least Privilege** network rules.

8.3.3 Right to Audit and Verification

Core Concept

Establishing the legal and technical basis for verifying the security of vendors and their products.

- **Right to Audit:** A contractual clause (Domain 3.8.3) that grants the consuming organization the legal right to inspect the vendor's code, processes, or operational environment.
    - *Use Case:* Essential for high-assurance or regulated systems where relying solely on a vendor's **SOC 2 Report** (Domain 8.2) is insufficient. It is a powerful **deterrent** and ensures independent verification.
- **Verification:** The process of ensuring that all the integrity measures (hashes, signatures) are checked at every security gate in the deployment pipeline.
    - *Implementation:* The **Application Security Toolchain** (Domain 7.3.2) must include steps to verify the digital signature and cryptographic hash of every component *before* it is integrated into the final build.

Domain 8.4 - Ensure and Verify Supplier Security Requirements

Definition

The implementation of contractual and technical controls during the vendor selection and acquisition process to ensure that external suppliers adhere to the organization's security standards before they are allowed to provide software or services. This is a critical risk mitigation step in the supply chain lifecycle.

8.4.1 Supplier Security Assurance and Verification

Core Concept

Moving beyond simple trust, the organization must verify that the supplier's internal security processes and infrastructure are robust enough to protect the provided components.

- **Audit of Security Policy Compliance (Secure Software Development Practices):** The process of verifying that the supplier follows its own documented secure development lifecycle (SSDLC).
    - *Implementation:* The customer sends a **security questionnaire** (e.g., based on the **Cloud Controls Matrix (CCM)** or **NIST SP 800-53**) asking for proof that the vendor performs **Threat Modeling** (Domain 4.4), **SAST/DAST** (Domain 6.2), and **security code reviews** during their development process.

    - *Verification:* Utilizing the contractual **Right to Audit** (Domain 8.3.3) to hire a third party to validate the vendor's claims, especially for high-risk components.

- **Security Track Record:** Reviewing the supplier's historical performance on security, including past breaches, vulnerability disclosures, and their **Average Remediation Time (ART)** (Domain 2.5) for critical flaws. A weak track record significantly increases the **Business Risk** (Domain 2.8).
- **Scope of Testing (Shared Responsibility Model):** Clearly defining the boundary of security responsibilities, particularly in cloud environments (SaaS, PaaS, IaaS).
    - *Implementation:* The agreement must specify whether the vendor is responsible for OS patching (PaaS) or if the customer is responsible for **application-level patching** and **Input Validation** (SaaS). This determines the scope of the customer's **Verification and Validation (V&V)** testing (Domain 6.8).

8.4.2 Incident Response and Monitoring Capabilities

Core Concept

Contractually ensuring that the supplier can effectively detect and respond to a security incident involving the components they provide, and that the customer receives timely, actionable information.

- **Vulnerability/Incident Notification, Response, Coordination, and Reporting:** The supplier agreement (SLA/MSA) must mandate:
    - **Timely Notification:** The maximum allowable time (e.g., 72 hours) for the supplier to notify the customer after a vulnerability or incident has been confirmed.
    - **Cooperation:** The requirement for the supplier to coordinate their **Incident Response Plan (IRP)** (Domain 2.9.2) with the customer's IRP, sharing forensic data and remediation steps.
    - **Reporting:** The format and frequency of reports detailing the status of the incident, root cause, and remediation status.
- **Log Integration into Security Information and Event Management (SIEM):** Requiring the vendor's service to provide a secure, standardized **log interface** that allows the customer to ingest security events into their own **SIEM** system (Domain 7.7.5).
    - *Purpose:* This is vital for **continuous monitoring** and allows the customer to correlate external vendor events with their internal application activity for real-time threat detection and forensic analysis.

8.4.3 Maintenance and Support Structure

Core Concept

Assessing the longevity, reliability, and security commitment tied to the component's support lifecycle, which impacts the long-term operational risk.

- **Maintenance and Support Structure (e.g., community versus commercial, licensing):** Reviewing who is responsible for providing security patches and updates.
  - **Commercial Support:** Provides guaranteed **Service-Level Agreements (SLAs)** (Domain 7.13) for patch delivery (lowering risk).
  - **Community Support:** For open-source components, support is voluntary and unpredictable, meaning the customer may need to implement **virtual patching** or accept the risk if no patch is available (higher risk).
- **End-of-Life (EOL) Policy:** Requiring the supplier to provide a formal EOL policy and advance notice when they plan to retire a product or service. This allows the customer to avoid the risk associated with running unsupported software (Domain 2.6).

Domain 8.5 - Support Contractual Requirements

Definition

The process of ensuring that all security commitments, liabilities, ownership rights, and operational expectations concerning third-party components or services are formally defined and legally binding within contracts and agreements.

8.5.1 Legal and Intellectual Property (IP) Clauses

Core Concept

Contracts must explicitly define ownership of custom-developed code and protect the organization's proprietary information when engaging a vendor.

- **Intellectual Property (IP) Ownership:** The contract must clearly state which party owns the IP rights to the code developed during the engagement. This is especially critical for **custom development** projects to ensure the customer retains full rights to the finished product.
- **Code Escrow:** A contractual agreement where a vendor's source code is held by a neutral third-party escrow agent.
  - *Purpose:* If the vendor goes bankrupt or fails to meet contractual obligations (e.g., stops providing security patches), the customer can legally access the

    - source code from the escrow agent to maintain and patch the software themselves. This protects the customer's long-term **Availability** (Domain 1.1.3).

- **Licensing (End-User License Agreement - EULA):** The EULA defines the terms under which the end-user can utilize the software, often placing restrictions on use (e.g., non-reverse engineering clauses). These terms must be enforced by the application's design (e.g., using **obfuscation** as a deterrent, Domain 5.1.5).

8.5.2 Liability and Assurance Clauses

Core Concept

Defining financial responsibility for security incidents and guaranteeing a minimum level of product safety.

- **Liability:** Contractual clauses that assign financial responsibility for damages, losses, or fines resulting from a security failure or breach caused by the vendor's software or service.
- **Warranty:** The vendor's guarantee that the software is free from material defects and often includes a commitment to fix security vulnerabilities discovered within a specified time frame.
- **Service-Level Agreements (SLA):** Legally binding agreements that define measurable operational security metrics the vendor must meet (Domain 7.13).
    - *Security Metrics:* Includes minimum **Availability** (uptime), maximum **Incident Response Time**, and guaranteed **patch delivery** speed for critical vulnerabilities. Failure to meet these thresholds often results in financial penalties.

8.5.3 Operational Integration and Exit Planning

Core Concept

Ensuring that the legal terms support the application's operational needs, particularly in maintaining continuity and managing separation.

Statement of Work (SOW) / Master Service Agreement (MSA): These documents must contain clauses explicitly mandating secure coding practices, vulnerability reporting procedures, and adherence to the customer's **Secure Coding Standards** (Domain 5.1).

- **Data Destruction Guarantee:** A legal requirement for the vendor to provide verifiable proof of data destruction upon contract termination, adhering to the customer's **Data Retention Policy** (Domain 3.4.4.1). This is critical for privacy compliance.
- **Access Revocation:** The contract should require the vendor to cooperate fully and immediately in revoking all their access and credentials upon contract termination or incident containment (Domain 2.6.1.1).