OWASP Secure-By-Design Framework

Draft Version 0.5.0 – Initial Community Review (August 2025)

Index

About OWASP

The Open Worldwide Application Security Project (OWASP) is a nonprofit foundation dedicated to improving the security of software. OWASP is community-led, vendor-neutral, and open. All of its projects, tools, and materials are free and open-source, created by an international team of practitioners, researchers, and volunteers.

The OWASP Foundation provides widely adopted standards, documentation, tools, and events to help organizations and individuals improve their application security. Learn more at https://owasp.org.

1 Secure by Design Framework

Secure-by-Design (SbD) is a foundational approach in the software development lifecycle that ensures security is engineered into applications and services from the outset, making them resilient to threats and aligned with both regulatory and organizational policies.

This framework delivers structured, actionable guidance for embedding security during the architecture and system **design phase** of the SDLC—long before code is written. It closes the gap between high-level security requirements and code-level verification standards, enabling teams to design systems where security is a built-in foundation rather than an afterthought.

By applying these practices early, you not only reduce vulnerabilities and costly rework, but also create systems that are inherently more reliable, maintainable, and easier to verify.

1.1 Scope

This framework focuses exclusively on **design-time decisions**—the architectural and systemic choices that determine how security is built into a solution. It is intended for:

- **System and product architects** who own solution design
- **Product engineers** evolving architecture through agile iterations
- **Security engineers** who review or advise during design

It does **not** cover secure coding practices (e.g., OWASP Top 10, ASVS), implementation-phase testing or scanning, or the threat-modeling methodology itself, which follows design. The guidance here is about shaping secure architecture before development begins, ensuring that implementation work starts from a secure foundation.

1.2 Project Deliverables

- **Structured Secure-by-Design Framework** – a comprehensive, principle-driven guide for design-phase security.
- **Design-Phase Security Checklist** – actionable review tool for architects and engineers to validate designs.
- **Best-Practice Guides** – covering microservices, resilience, service-to-service interactions, and security architecture patterns.
- **Secure API & Messaging Guidance** – practical recommendations for HTTP, gRPC, Kafka, AMQP, and event-driven designs.
- **Reference Implementations & Real-World Examples** – illustrating how principles are applied in practice.
- **OWASP-Hosted Presentations & Training Materials**

1.3 Why This Project Matters

Security flaws introduced during system design are often the most costly and complex to remediate—sometimes requiring fundamental architectural changes that ripple through the entire product. By embedding **Secure-by-Design** practices early, we prevent these issues before they materialize, avoiding expensive rework and reducing risk exposure.

Modern systems are increasingly distributed, API-driven, and interconnected. This complexity amplifies potential attack surfaces, dependency risks, and failure modes. Without intentional, principle-based design, security controls become inconsistent, bolt-on, and fragile—leaving gaps that can be exploited.

This project provides a structured, repeatable, and scalable way to align architecture decisions with proven security principles. It empowers:

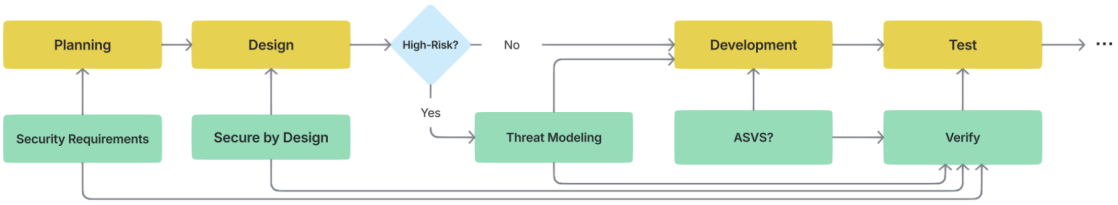
- **Architects and engineers** to make informed design choices with security built in.
- **Security teams** to engage earlier, influencing system shape rather than reacting to flaws.
- **Organizations** to meet regulatory and customer trust expectations without slowing delivery.

In short, it shifts security from reactive gatekeeping to proactive enablement, making it part of the natural flow of product design and evolution.

1.4 Relationship to Other OWASP Projects

Secure-by-Design does not exist in isolation—it is a critical link between other OWASP security practices across the entire SDLC. As shown in the diagram, the process begins in the **Planning** phase, where formal security requirements are established (e.g., via OWASP ASVS requirements mapping). These define the baseline protections the system must meet. In the **Design** phase, the SbD framework applies structured principles, architectural patterns, and specific controls to address those requirements before any code is written. For high-risk or business-critical projects, an optional **Threat Modeling** checkpoint follows to validate that chosen SbD controls mitigate the most critical threats and leave no significant risks unaddressed.

The lifecycle then flows into **Development**, where teams follow OWASP ASVS and other secure-coding standards to ensure implementation faithfully reflects the secure design. In the **Test** phase, security verification confirms that all layers—design, code, configuration, and deployment—work together as intended before release. This alignment embeds security at every stage, with SbD providing the architectural backbone that other OWASP practices reinforce. The result: systems that are secure by construction, not by retrofit.



End-to-End Security Integration in the SDLC

1.4.1 Security Requirements

- **Scope:** Defined at the earliest possible stage (Planning) and tracked throughout the lifecycle.
- **Comparison:** Security requirements define *what* protections the system must deliver; Secure-by-Design prescribes *how* to embed those protections into the architecture.
- **How They Work Together:** Security requirements set explicit objectives (e.g., confidentiality, integrity, availability, privacy, compliance). SbD provides a disciplined method to translate each requirement into one or more architectural controls, ensuring every requirement is addressed in the blueprint before code is written.
- **Focus:** Requirements specify the “must-haves” (e.g., MFA, data encryption, retention limits). SbD maps these to concrete design elements and vetted patterns—such as least-privilege zones, encrypted channels, and circuit breakers—so the protections are enforced at an architectural level.
- **Outcome:**
    - **Security Requirements:** A vetted, testable list of statements.
    - **SbD:** A hardened architecture—data-flow and component diagrams annotated with controls—ready for secure implementation.
- **Methodology:**
    - **Security Requirements:** Elicit from stakeholders (business, legal, compliance, operations), map from regulations (GDPR, PCI-DSS) and standards, incorporate results of risk/threat analyses, and rank by impact. Update whenever designs change or new risks emerge.
    - **SbD:** Apply core principles (least privilege, defense in depth, secure defaults), select vetted patterns, and validate using a design-time checklist to ensure coverage of all requirements.
- **Documentation:**
    - **Security Requirements:** Captured in a formal Security Requirements Specification (SRS) and compliance/risk traceability matrix.
    - **SbD Artifacts:** Architecture Decision Records (ADRs), annotated diagrams referencing specific patterns, and completed SbD checklists.
- **Example:**
    - **Security Requirement:** “All users must authenticate with at least two factors (password + OTP) before accessing any dashboard or API”.
- **SbD Implementation:**

- Place an OAuth2/OpenID Connect Identity Provider (IdP) in front of all services.
- Configure MFA enforcement at first login and every 30 days thereafter.
- Route all user logins through an API Gateway that validates and forwards JWT tokens to downstream services.
- Document the IdP trust boundary in architecture diagrams; no service accepts credentials directly.

1.4.2 OWASP Threat Modeling

- **Scope:** Conducted after an initial design is complete, focusing on post-design risk analysis.
- **Comparison:** Threat Modeling assumes you already have an architecture to analyze; SbD creates that architecture with embedded controls from the outset.
- **How They Work Together:** By applying SbD first, threat modeling becomes faster and more targeted—identifying edge-case scenarios and advanced threats rather than exposing fundamental design flaws that require costly redesigns.
- **Purpose:**
  - **SbD:** Integrates security principles and proven patterns during design to proactively prevent classes of vulnerabilities.
  - **Threat Modeling:** Evaluates an existing design from an attacker's perspective, identifying potential threats and determining mitigation strategies.
- **Focus:**
  - **SbD:** Principle- and pattern-driven, emphasizing service isolation, least privilege, data protection, and system resilience.
  - **Threat Modeling:** Risk- and adversary-driven, focusing on attack vectors, likelihood/impact analysis, and specific threat mitigations.
- **Outcome:**
  - **SbD:** Hardened architecture with built-in controls aligned to policies and standards.
  - **Threat Modeling:** Threat models, risk assessments, and updated mitigation plans for the implemented design.
- **Methodology:**
  - **SbD:** Apply principles like least privilege and secure defaults as part of architecture creation.
  - **Threat Modeling:** Use techniques such as STRIDE, DREAD, and attack trees to systematically identify and evaluate threats.
- **Documentation:**
  - **SbD:** Architecture diagrams, design specifications, and security requirement mappings.
  - **Threat Modeling:** Threat model diagrams, risk registers, and revised security requirements.
- **Examples:**
  - **SbD:** Define clear service boundaries, enforce TLS/mTLS, design resilient failure modes.

- **Threat Modeling:** Analyze authentication workflows, assess third-party integrations, trace sensitive data flows for potential exposure points.

1.4.3 OWASP ASVS

- **Scope:** ASVS is a secure-coding and verification standard applied primarily during the Development and Testing phases to ensure implementations meet agreed-upon security controls.
- **Comparison:**
  - **SbD:** Focuses on *design-time*—embedding security into architecture before code is written.
  - **ASVS:** Focuses on *implementation-time*—verifying that the code, configuration, and deployed system meet specific security requirements.
- **How They Work Together:**
  - SbD produces a hardened blueprint that specifies which controls must be implemented. ASVS provides the verification criteria to confirm those controls are correctly implemented and effective in the running system.
- **Purpose:**
  - **SbD:** Prevents security gaps by addressing them in architecture and design.
  - **ASVS:** Detects implementation gaps or misconfigurations through structured verification.
- **Focus:**
  - **SbD:** High-level design controls (e.g., service isolation, mTLS enforcement, input validation at the gateway).
  - **ASVS:** Detailed verification of implementation aspects (e.g., code correctly validates tokens, TLS configuration resists downgrade attacks, input sanitization is robust).
- **Outcome:**
  - **SbD:** Annotated architecture and design artifacts with mapped security controls.
  - **ASVS:** A verified application meeting security control requirements at the implementation level.
- **Methodology:**
  - **SbD:** Apply design principles and patterns; capture decisions in ADRs and diagrams.
  - **ASVS:** Test and review the application against ASVS's tiered verification levels (V1–V14 categories). Use automated scanning where applicable but rely heavily on manual verification for logic and security workflows.
- **Documentation:**
  - **SbD:** ADRs, architecture diagrams, control-to-requirement mapping.
  - **ASVS:** Verification reports, test results, defect logs, and remediation tracking.
- **Examples:**
  - **SbD:** Specify that all sensitive data at rest must be encrypted using AES-256 and managed keys in KMS.

- **ASVS:** Verify that the database schema uses encrypted storage, the KMS integration is correctly configured, keys are rotated according to policy, and no sensitive data is stored unencrypted in logs or caches.

1.5 Key Differentiators

- **Proactive by Design**: Embeds controls before code is written.
- **Pattern-Centric**: Goes beyond abstract principles by providing concrete, reusable patterns for microservices, APIs, messaging, and integration scenarios.
- **Checklist-Driven**: Offers a concise (≤ 40-item) control-oriented design checklist to enable fast, consistent reviews across teams.
- **Extensible**: Built for growth, with a modular structure that supports community contributions, reference architectures, and integration with future automation or tooling.

1.5 Key Differentiators

- **Proactive by Design**: Embeds controls before code is written.
- **Pattern-Centric**: Goes beyond abstract principles by providing concrete, reusable patterns for microservices, APIs, messaging, and integration scenarios.
- **Checklist-Driven**: Offers a concise (≤ 40-item) control-oriented design checklist to enable fast, consistent reviews across teams.
- **Extensible**: Built for growth, with a modular structure that supports community contributions, reference architectures, and integration with future automation or tooling.

1.6 What Is a Secure-by-Design Review?

A **Secure-by-Design (SbD) Review** is a structured assessment of an application's architecture and design, performed *before development begins*, to ensure that security is engineered into the system from the outset.
It systematically examines components, data flows, interfaces, dependencies, and trust boundaries to verify that required security controls are built directly into the architectural blueprint.

Typical outcomes include:

- Annotated architecture and data-flow diagrams clearly showing applied controls, trust zones, and security-relevant boundaries.
- A completed SbD review checklist (see Section 4) summarizing implemented controls, residual risks, and gaps.
- Architecture Decision Records (ADRs) and a prioritized set of action items to address identified gaps.
- Risk-based recommendation on whether to run a Threat Modeling exercise before development begins.

1.7 When and How Should It Be Considered?

SbD Reviews should be performed early and iteratively—during the Planning and Design phases, at the start of major epics, and whenever there is a significant architectural change (e.g., new external exposure, introduction of sensitive data, adoption of novel technology, or Tier-1 impact).

Run an SbD Review whenever:

- The architecture introduces new trust boundaries or external integrations.
- The system will handle regulated or sensitive data.
- The change could materially affect resilience, scalability, or security posture.

The **review process**, participant roles, and hand-off mechanics are described in **Section 2 – Secure-by-Design Process**.

1.8 Expected Benefits

By applying this framework and running Secure-by-Design reviews, teams can:

- **Enhance Security Posture** – Reduce attack surface and eliminate entire classes of flaws early.
- **Ensure Compliance** – Align architectures with legal, regulatory, and internal security requirements from the outset.
- **Improve Reliability** – Design systems that are robust, scalable, and able to degrade gracefully under failure conditions.
- **Facilitate Collaboration** – Use the SbD Review Checklist as a concise hand-off artifact for rapid, targeted AppSec feedback.
- **Streamline Development** – Identify issues during design to minimize costly rework; produce auditable artifacts (diagrams, ADRs, checklists, threat models) with clear traceability from requirements to implemented controls.

2 Secure-by-Design Process

This section provides the operational playbook for applying Secure-by-Design during the design phase and engaging with AppSec efficiently—without creating bottlenecks. It describes the end-to-end process, what a design review examines, when to escalate, how the SbD Review Checklist fits into the workflow, and the mechanics for handing off to AppSec.

Two complementary tools underpin the process:

- **SbD Principles & Recommendations** – The full guidance across all domains (architecture, data, resilience, access control, monitoring). Used actively during design. *(See Section 3).*
- **SbD Review Checklist** – A concise Yes/No/N-A questionnaire capturing the most critical controls, used for self-review and as the communication bridge to AppSec. *(See Section 4).*

Each team designates a **Security Champion** as the primary security contact. They own the checklist, coordinate any required AppSec engagement, and ensure that design artifacts remain complete, accurate, and in sync with the implemented architecture.

2.1 Process Overview

1. **Capture Security Requirements (Planning)**
    - Define *what must be true* (CIA, privacy, compliance) in the SRS/backlog with traceability.
2. **Draft High-Level Architecture (Design)**
    - Sketch components, trust zones, data flows, external dependencies, and assumptions.
3. **Apply SbD Principles & Patterns**
    - Select patterns (service isolation, mTLS, idempotency, circuit breakers, schema governance) that satisfy the requirements.
    - Consider the recommendations detailed in Section 3.
    - *Security Champion involved for early alignment.*
4. **Complete the SbD Review Checklist**
    - Answer Yes/No/N-A for each control and add comments/links to ADRs, diagrams, and policies. (See section 4)
    - *Security Champion facilitates completeness and clarity.*
5. **Internal Peer Review**
    - Another architect/engineer reviews the design + checklist; unresolved “No” items become actions.
6. **Risk Triage**
    - Low / Normal Risk → Proceed to development (step 8).
    - High / Critical Risk → Ping AppSec (attach the checklist + security requirements + diagrams).

    - *Security Champion* — acts as the point of contact with AppSec.
7. **Optional Threat Modeling & AppSec Review (when triggered)**
    - If risk triggers are met, the product team performs a lightweight or full threat model (methods like STRIDE) before development, with AppSec support if needed.
    - AppSec reviews the checklist and threat model, focusing on “No”/“N-A” items with high impact, and recommends mitigations.
8. **Finalize Design Artifacts**
    - Update diagrams, ADRs, checklist status, and (if applicable) threat model.
9. **Handoff to Development**
    - Ensure the design and checklist are part of the dev onboarding pack.
10. **Design-Drift Watch**
    - For any major change (new data type, new exposure, new tech), repeat steps 3–8.

2.2 What the Secure-by-Design Review Covers

A Secure-by-Design review evaluates the architecture and high-level design of a system against a curated set of domains from the SbD Principles & Recommendations (Section 3).

The review inspects the following domains:

- **Core Secure Design Principles:** Confirmation that foundational principles—such as least privilege, defense in depth, secure defaults, and zero-trust boundaries—are explicitly applied in the architecture.
- **Architecture & Service Design:** Assessment of service boundaries, trust zones, inter-service communication methods, integration points, and scalability/resilience patterns.
- **Data Management & Protection:** Verification of data classification, ownership, encryption (in transit/at rest), retention/deletion policies, and auditability mechanisms.
- **Reliability & Resilience:** Evaluation of fault tolerance, error handling, failover, backoff/retry policies, bulkhead and circuit-breaker patterns, and performance safeguards.
- **Access Control & Secure Communication:** Review of authentication and authorization approaches (user/service), secure channel enforcement (TLS/mTLS), secret/key management, and policy enforcement mechanisms.
- **Monitoring, Testing & Incident Readiness:** Confirmation that observability, logging, metrics, and incident response capabilities are designed in from the start, with sufficient detail to detect, investigate, and recover from security events.

2.3 Risk-Based Escalation & Threat Modeling

The Secure-by-Design process is designed to be lightweight for most changes, but specific risk triggers require escalation to a Threat Modelin**g** activity—either facilitated by AppSec or

performed with their guidance. Threat Modeling provides a deeper, attacker-oriented review to validate that the proposed architecture's controls adequately mitigate the most critical risks.

Escalation Triggers (any one of these should prompt a threat model before development begins):

- **Sensitive or Regulated Data:** Handling personally identifiable information (PII), payment card data (PCI), health records (PHI), or other regulated datasets.
- **New External Exposure:** Introducing new public-facing interfaces—such as APIs, message endpoints, or event streams—that may require abuse-case and misuse-case analysis.
- **Novel Technologies or Patterns:** Adopting new frameworks, infrastructure components, or architectural patterns (e.g., a message broker not previously used, a new orchestration platform, or emerging protocols) that lack established security baselines in your environment.
- **High-Business-Impact Services (Tier-1):** Delivering or significantly modifying critical services whose compromise would have severe operational, financial, or reputational consequences.

When triggered, the Threat Modeling session:

- Starts with the current SbD-hardened architecture as input,
- Analyzes potential attack vectors and abuse scenarios,
- Validates that existing controls sufficiently reduce risk,
- Identifies additional mitigations if needed.

Outputs typically include an updated threat model diagram, a prioritized risk register, and action items to address uncovered gaps—feeding back into the SbD artifacts (ADRs, diagrams, checklists).

2.4 SbD Review Checklist — Usage

It's a quick, control-oriented snapshot that communicates what's in place and what needs work, enabling fast security team triage. Refer to Section 4, for more details.

Each row includes: *(a)* control name, *(b)* Yes/No/N-A, *(c)* comment "how/where implemented" with links (ADR-IDs, diagram anchors, policy refs), *(d)* evidence link (optional).

When to complete.

- Before any AppSec request.
- On major design change (new trust boundary, sensitive data, new external exposure, novel tech).

3 Secure-by-Design Principles & Recommendations

This section provides the authoritative design-time guidance for building systems that are secure by construction. It does not cover coding standards such as those in the ASVS, the use of automated scanners, or vulnerability triage processes. Instead, it provides architecture-level controls aimed at eliminating entire classes of flaws—such as through the application of least privilege, strong isolation, idempotency, disciplined schema management, and mutual TLS (mTLS).

The guidance is organized into six domains as follows which translates the project's security requirements into concrete, architect-level controls, and provides actionable, design-time recommendations that architects and product teams apply while shaping systems.

How to Use:

- Start at the domain relevant to your current design decision and record decisions as ADRs and on architecture/data-flow diagrams.
- For each recommendation: review the Rationale, apply the Control, consider the Implementation Notes, and capture Evidence.
- Use the SbD Review Checklist (section 4.3) to summarize what you've applied (Yes/No/N-A + justification) for internal review and to communicate with AppSec when needed.
- For concrete, end-to-end examples that implement these controls, see section 5's Reference Architectures & Patterns.

Domain 1 - Core Secure Design Principles

- **Least Privilege** — Minimize permissions for users, services, pipelines, and tools; deny-by-default; narrowly scope tokens/roles; time-bound access.
- **Defense in Depth** — Layer network isolation, authN/Z, input validation, encryption, rate-limiting, monitoring, and alerting so one failure isn't fatal.
- **Secure Defaults** — Enable TLS, private networking, strict security headers, safe cipher suites, and hardened baselines by default.
- **Zero-Trust & Explicit Boundaries** — Treat all networks and callers as untrusted; authenticate and authorize every call; make trust zones explicit on diagrams.
- **Fail Secure / Graceful Degradation** — Prefer safe failure (deny) to insecure success; avoid leaking internals in errors; design partial modes.
- **Simplicity & Minimized Attack Surface** — Reduce components, open ports, conditional paths; favor small, cohesive interfaces over sprawling ones.
- **Observability by Design** — Emit structured logs, traces, and security events with correlation/trace IDs; standardize logging libraries.
- **Contract-First & Versioned Interfaces** — Define OpenAPI/AsyncAPI/Avro/Protobuf up front; validate at edges; version breaking changes.

- **Automation & Repeatability** — Use IaC and policy-as-code; template security configurations; enforce in CI (linting, drift checks).

Domain 2 - Architecture & Service Design

Service Design & Boundaries

- Align services to domain boundaries (DDD); each owns its data and business logic—others use its API/events. Avoid shared write databases.
- Eliminate circular dependencies; refactor to reduce coupling; prefer pub/sub over chatty RPC to reduce tight coupling.
- Keep services right-sized (cohesive, independently deployable); avoid nano-services and accidental monoliths.
- Document dependencies, SLOs, and data ownership in READMEs/runbooks.
- Centralize shared business logic where appropriate (avoid duplication; prevent inconsistent enforcement).
- Document dependencies, SLOs, and data ownership in READMEs/runbooks so on-call knows what breaks what.

Trust Zones / Network Isolation

- Place workloads in isolated trust zones; allow direct access only within a zone.
- Force cross-zone access via Integration Services (API gateway/mesh/bus) where policies are enforced and traffic is observable.
- Use a unified addressing/DNS scheme for external references; intra-zone shortcuts are OK but don’t leak outside.
- Keep addressing and data-access methods consistent across environments to reduce “works in dev only” issues.

Cross-Zone Integration Services

- Route HTTP/gRPC through an API gateway: enforce authN/Z, rate-limits, schema validation, and request logging.
- Use a message bus (Kafka/AMQP/Redis streams) for async cross-zone communication; centralize policies, ACLs, and auditing.
- Treat Integration Services as shared, governed assets with strict access controls and owner approval.

Inter-Service Communication

- Use a service mesh with mTLS for identity + encryption between services; it removes the need to manage certs in each app.
- Standardize client libraries for retries, timeouts, circuit-breakers, and idempotency.

- Avoid long synchronous chains across many services (N services in a line). Break chains with queues/events to absorb spikes and failures.

Messaging Infrastructure

- Choose log-based (ex., Kafka-class) vs message-based (ex., AMQP/Redis) deliberately; document the trade-off (replay vs simplicity).
- Configure durability (persistence, replication, backups) and topic/queue retention by data sensitivity.
- Monitor throughput, lag, latency, and resource usage; plan capacity/partitioning ahead of load.
- - Define DLQ behavior to route messages that cannot be processed after retries; and monitor and alert on growth; have a triage/replay playbook workflow.

API & Event Design / Schema Management

- Separate internal vs global schemas/topics; establish enterprise naming (e.g., OrderCreated_v1).
- Specify required/optional fields and constraints; standardize on JSON/Avro/Protobuf so tooling works everywhere.
- Plan backward/forward compatibility; version breaking changes; publish deprecation timelines.
- Validate and sanitize all inbound inputs at the first hop; reject unknown fields; use approved libraries.
- Design events to be multi-consumer safe (self-contained context, idempotent consumption).
- Notify impacted consumers on breaking changes via an agreed process (owners list, subscriptions, or catalogs).

Legacy Systems Integration

- Add an anti-corruption layer to translate old models/protocols; keep legacy assumptions from leaking into new services.
- Replace legacy components incrementally (strangler pattern) with measurable milestones.

Scalability & Startup Resilience

- Design for horizontal scale; set autoscaling triggers and a minimum replica count (so you don’t scale to zero by mistake).
- Start safely when dependencies aren’t ready: timeouts, retries with backoff, and feature flags.
- Warm caches or precompute hot data to reduce cold-start latency.

Service Discovery

- Centralize discovery (DNS/mesh registry) and pin service identities (SPIFFE/SVID-class in meshes) to know who is who.
- Publish service contracts in catalogs (OpenAPI/AsyncAPI) so teams can find and integrate safely.

Domain 3 - Data Management & Protection

Data Ownership & Domain Modeling

- Each service must own its data store (write access). Others read via API or subscribed events. Prevents tight coupling and “stealth joins”.
- Maintain data dictionaries and owners; update when schemas evolve.

Idempotency

- Make handlers idempotent; tolerate retries and duplicates (repeated deliveries) without side-effects.
- Use stable identifiers and store processed IDs/checksums to suppress duplicates.
- Provide exactly-once effects through idempotent operations + dedupe, not broker magic.

Transaction Management

- Prefer local ACID transactions within a service where possible to avoid partial updates before cross-service workflows.
- For multi-service workflows or long-running transactions, use Sagas (orchestration/choreography).
- Avoid distributed transactions such as 2-phase commit (2PC)—it’s complex and brittle at scale.
- Define compensating actions (for rollback) next to the happy path.

Data Consistency

- Embrace eventual consistency; document staleness windows and reconciliation jobs.
- Design UI/UX to show “processing”/“pending” states and offer retry or manual resolve.
- Consider event sourcing when you need full history/audit or state reconstruction.

Data Classification & Protection

- Classify data (public/internal/confidential/restricted) with a named owner; review periodically.

- Apply appropriate security controls based on data classification (e.g., encryption for sensitive, RBAC for confidential/restricted).
- Encrypt in transit and at rest using approved algorithms; centralize key management/rotation; separate duties.
- Minimize data collection to business necessity; document rationale/lawful basis where applicable.

Data Retention & Deletion Policies

- Set retention periods \+ secure deletion/archival for data class; meet regulatory/contractual obligations.
- Periodically purge unnecessary data; verify deletion propagates to backups/replicas.

Environment-Consistent Data Access Methods

- Standardize connection methods and secrets retrieval across environments.
- Avoid environment-specific quirks that break portability and security.

Auditability & Data Lineage

- Record who/what/when for sensitive data changes; Track lineage from source to sink.
- Log schema changes and approvals; catalog datasets and owners.

Domain 4 - Reliability & Resilience

Error Handling

- Use retries with exponential backoff \+ jitter; cap attempts; avoid retry storms.
- Return safe, standardized errors (no stack traces or secrets). Document error models.

Asynchronous Communication Semantics

- Decide ordering requirements; use keys/partitions or total ordering where needed.
- Select delivery semantics (at-least-once vs exactly-once trade-offs); document choices.
- Set ack/retry/visibility timeouts per queue/topic; define DLQ policies and a replay process.
- Design Integration Services for high availability and cross-zone error handling so shared layers don’t propagate failures.
- Explicitly accept and design for timing delays that may affect outcomes (latency/timeout windows).

Isolation Patterns

- Apply circuit breakers to failing dependencies to prevent cascading failures when a dependent service is down; bulkheads to isolate pools/threads so one spike doesn't sink the ship.
- Provide fallbacks (cached data, default responses) or degrade non-critical features instead of cascading failure.

Race Conditions & Concurrency Controls

- Use distributed locks (e.g., Redis-class) or atomic ops for shared-state updates.
- Require Idempotency-Key for all mutating endpoints; on same key+payload → return stored response; same key+different payload → 409 Conflict.
- Combine with rate-limits and DB uniqueness/optimistic concurrency to handle concurrent modifications safely within a service..

Timeouts, Health Checks & Service Availability

- Set timeouts for all calls; include liveness/readiness probes; enable auto-restart.
- Design degraded modes when dependencies are down; prefer partial functionality (queue-and-replay, read-only, feature flags).
- Define explicit failover/redundancy (active-passive, multi-AZ/region) strategies for critical paths.

Scalability, Resource Quotas & Rate Limiting

- Define CPU/mem/IO quotas and autoscaling policies; prevent noisy-neighbor issues.
- Enforce gateway/mesh rate limits (per endpoint, per client) with bursts and sensible defaults.

Performance Optimization & Caching

- Cache on client/service/CDN for hot paths; define eviction and invalidation rules.
- Apply back-pressure and queue sizing for spikes; shed load gracefully. Set minimum quotas so the service can operate even under platform caps (in case, platform configurations may limit resource usage to prevent overconsumption by a single service).

Dependency & Third-Party Risk Management

- Evaluate external services/OSS (license, support, vulnerability posture); follow onboarding policy.
- Track components; monitor for CVEs; pin versions; plan exit strategies.

Deployment & Release Management

- Prefer canary/blue-green; set guardrail metrics; test rollback paths regularly.
- Use feature flags for progressive delivery and kill switches for fast disable.

Decommissioning & Disposal

- Retire DNS, CI/CD, storage, credentials; wipe data securely; verify no residual access.

Supply Chain Integrity & Artifact Provenance

- Use signed, scanned artifacts from trusted registries; enforce provenance checks in CI.
- Generate/retain SBOMs (e.g., CycloneDX); gate builds on policy compliance.

Domain 5 - Access Control & Secure Communication

Secure Communication & Service Identity

- Enforce TLS/mTLS for all service communications; automate cert issuance/rotation (mesh SDS-style).
- Pin service identities (SPIFFE/SVID-class) for workload-to-workload trust (to prove who is calling whom).

Authentication

- Centralize user auth with OIDC/OAuth2 IdP; enforce MFA for privileged paths.
- Prefer short-lived tokens and workload identities over long-lived static secrets.

Authorization & Permissions

- Implement RBAC/ABAC; document "who/what can call which APIs or publish/consume events".
- Centralize authorization policies (gateway/mesh where possible) using policy-as-code for consistency.
- Govern shared Integration Services with strict access lists (ACLs) and approvals.
- Externalize business rules (configs/policies) instead of hard-coding.
- Periodically review and right-size permissions for users, services, pipelines, tools.

Configuration Security & Secret/Key Management

- Store secrets in a secret manager; rotate keys/certs regularly; audit access. Avoid secrets in env vars/logs.

- Use dynamic/ephemeral credentials when available.

Compliance & Regulatory Requirements

- Map controls to GDPR/PCI DSS/etc; document compensating controls where needed.
- Maintain evidence (policies, DPA, DPIA) as part of design artifacts.

Identity & Access Management

- Define roles and separation of duties; protect log/admin path; least privilege everywhere.

Least Privilege for Tooling & Pipelines

- Restrict CI/CD, VCS apps, chat-ops, and integrations to minimal scopes; require approvals for sensitive actions.
- Protect signing keys/runners; isolate build environments.

Domain 6 - Monitoring & Incident Readiness

Logging & Observability

- Emit structured logs with correlation/trace IDs; centralize ingestion and retention.
- Log administrative and operational access; restrict access; protect logs from tampering and over-exposure.

Metrics, Dashboards & SLOs

- Track latency, error rate, saturation, and key business KPIs. Build health dashboards.
- Monitor event flows (queue sizes, lag, DLQ volume); alert on anomalies.

Security Testing & Verification

- Plan ASVS-aligned verification; include negative tests from threat modeling.
- Validate SbD controls (isolation tests, mTLS handshake checks, authorization denials, rate-limit behavior).

Documentation & Knowledge Sharing

- Publish OpenAPI/AsyncAPI and event schemas; maintain catalogs.
- Keep runbooks and ADRs up to date; document data ownership and dependencies.

Incident Response & Recovery

- Maintain a formal Incident Response plan (roles, comms, evidence handling); rehearse with tabletops.
- Detect suspicious patterns: repeated failed logins, unusual 403 rates, anomalous access to sensitive endpoints, geo-improbable logins, unexpected config/file changes.
- Feed lessons learned back into security requirements, patterns, ADRs, and the checklist.

Audit & Retention

- Retain audit logs per policy (e.g., ≥12 months with a searchable window).
- Ensure tamper-evidence and chain-of-custody for security logs.

4. Secure-by-Design Review Checklist

The Secure-by-Design checklist set is a practical tool for embedding security requirements into the design phase of the software development lifecycle and for ensuring those requirements are verified before implementation. It serves as a clear, concise mechanism for product teams to self-assess their designs, capture evidence, and internally document their security posture—without introducing unnecessary delays. These checklists are intended primarily for internal team self-review and as a shared communication tool with the security team in high-risk projects.

Checklists translate the framework's principles and recommendations (Section 3) into actionable, reviewable controls. They ensure that security-critical aspects are consistently considered, documented, and validated across all projects, regardless of the team's size, experience, or technology stack. When used systematically, they reduce rework, strengthen the security posture, and enable efficient, scalable collaboration within the team and, where appropriate, with AppSec.

This section defines the structure, intended usage, and governance of Secure-by-Design checklists, clarifying how they fit into the design process and how they can be maintained alongside other project artefacts.

4.1 Intended Use and Timing

The SbD review checklist is completed by the product team during the design phase—ideally before high-level design sign-off or epic kickoff—and is refreshed whenever the design changes in a way that could alter the system's security characteristics. Typical triggers include new integrations, exposure to untrusted networks, handling of sensitive data, adoption of new technologies, or changes to core workflows.

Begin with the full checklist during detailed design sessions—either from the outset of design work or as a dedicated security refinement step—to systematically think through applicable recommendations listed in Section 3 and update the design accordingly. Capture your considerations and decisions either directly within the design artefacts or in a separate security notes document; for internal peer review, assurance, and historical record.

Once these detailed considerations have been addressed, complete the SbD review checklist to summarize them into a clear, high-level mapping against the security requirements identified earlier. This becomes the main descriptive summary of applied considerations, along with any gaps to address later, and is stored with the project's artefacts for ongoing reference. Where the project is high-risk, this document can also support discussions with the AppSec to request guidance or additional assurance. See section 4.4 for escalation and risk thresholds.

4.2 Checklist Structure

Each checklist entry contains:

- **ID** – Unique identifier (e.g., `AS-05`) for traceability.
- **Control Statement** – The security control or question to be addressed.
- **Status** – *Yes / No / N-A*.
- **Justification** – Stable links to ADRs, diagrams, configurations, relevant policy references, or including brief explanations or justifications where relevant.
- **Severity** – High, Medium, or Low if the control is not implemented.
- **Comment** – Supplementary notes for any status value, including clarification, additional context, or rationale for "N-A" and "No".

4.3 SbD Review Checklist

This checklist defines the minimum set of controls that every design is expected to address. A CSV version is available here. Each item abstracts key recommendations from Section 3, streamlined to support a Yes/No/N-A decision backed by evidence. For each control, teams record *Status*, *Justification*, *Severity*, and *Comments*. Items marked (Critical) are generally required to be Yes at design sign-off, or to have an approved, time-bound mitigation plan in place.

A. Architecture & Service Design

ID	Control Statement	Critical
AS-01	Trust zones (logical network "Spaces") are enforced; cross-zone traffic flows only via governed integration layers (gateway/bus).	Yes
AS-02	Service addressing & discovery are unified and consistent across environments; intra-zone shortcuts are not used externally.	

AS-03	Service boundaries are clear; no circular dependencies; services are right-sized; each service owns its data.	
AS-04	Architecture avoids long synchronous chains; event-driven or queued handoffs are used where appropriate.	
AS-05	API/event contracts are defined and versioned; consumers are notified before breaking changes take effect.	
AS-06	Messaging has durability (persistence/replication/retention) with a defined DLQ strategy and triage.	
AS-07	Startup resilience: services handle missing dependencies; autoscaling defined with minimum replicas.	
AS-08	Legacy integrations are isolated behind an anti-corruption layer; migration path is defined.	

B. Data Management & Protection

ID	Control Statement	Critical
DM-01	Data are classified with named owners; controls are proportional to classification.	
DM-02	Encryption in transit (TLS/mTLS) and at rest with managed keys and rotation is designed in.	Yes
DM-03	Handlers and workflows are idempotent; duplicate detection/suppression is in place where needed.	
DM-04	Cross-service transactions use Sagas/compensations; 2PC is avoided; prefer intra-service ACID first.	
DM-05	Retention/deletion policies exist per data class; minimization is applied to collection/storage.	
DM-06	Consistency model is documented; UX handles staleness/conflicts; event-sourcing considered where appropriate.	

C. Reliability & Resilience

ID	Control Statement	Critical
RR-01	Retries use exponential backoff + jitter; default exception handling and safe error models exist.	

RR-02	Circuit breakers/bulkheads protect dependencies; non-critical features have fallback/degraded modes.	
RR-03	Asynchronous semantics are defined: ordering, delivery guarantees, ack/visibility timeouts; DLQs are monitored.	
RR-04	Shared integration layers (gateway/bus) are high-availability and contain errors to prevent cross-zone failure propagation.	
RR-05	All mutating endpoints enforce an Idempotency-Key; concurrency controls (locks/optimistic concurrency) are in place.	
RR-06	Timeouts on all calls; health probes; explicit failover/multi-AZ strategies for critical paths.	Yes
RR-07	Quotas and autoscaling are defined; rate-limits enforced at edges; minimum capacity documented.	
RR-08	Caching/CDN/back-pressure patterns are applied where appropriate; performance guardrails are defined.	

D. Access Control & Secure Communication

ID	Control Statement	Critical
AC-01	All communications use TLS/mTLS; service identity is verified (e.g., mesh-issued certs).	Yes
AC-02	Central IdP (OIDC/OAuth2) is used; MFA enforced on privileged/admin paths; tokens are short-lived.	Yes
AC-03	RBAC/ABAC governs APIs and publish/consume on messaging; least privilege is applied.	
AC-04	Authorization is centralized at gateway/mesh via policy-as-code; shared integration services are governed.	
AC-05	Secrets are stored in a secret manager; keys/certs rotate automatically; no secrets in code/logs.	
AC-06	Applicable regulatory controls (e.g., privacy/financial) are identified with evidence of design alignment.	
AC-07	Least privilege extends to CI/CD, VCS apps, chat-ops, and third-party tooling; periodic reviews are scheduled.	

E. Monitoring, Testing & Incident Readiness

ID	Control Statement	Critical
MT-01	Structured centralized logs with correlation/trace IDs; administrative access is logged.	
MT-02	Metrics/SLOs and dashboards cover service health and event-flow anomalies; alerts are actionable.	
MT-03	ASVS-aligned security tests and negative tests from threat-modeling are planned in the test strategy.	
MT-04	SbD controls are verifiable in test (isolation, mTLS, authZ denials, rate-limit behavior).	
MT-05	OpenAPI/AsyncAPI and event catalogs are published; ADRs/runbooks are current.	
MT-06	A documented Incident Response plan (roles, comms, evidence handling) exists and is rehearsed.	Yes
MT-07	Audit log retention ≥12 months with tamper-evidence and a searchable hot window is defined.	Yes

4.4 Escalation & Risk Thresholds

If any critical control is No, or the overall risk score exceeds your team's threshold, engage AppSec before development proceeds. Critical controls typically include those related to trust boundaries, encryption in transit/at rest, authentication and authorization, and incident readiness.

Suggested scoring: Yes = 0, N-A = 0 (with justification), No (Low) = 1, No (Med) = 2, No (High) = 4.
Escalate when: any critical = No, or total score ≥ 6, or any trigger from section 2.4 applies (new external exposure, sensitive data, novel tech, Tier-1 impact).
Review scope: Security team reviews the checklist and linked evidence, advises on mitigations, and may request deeper verification or threat modeling.

4.5 Governance and Maintenance

Checklists are version-controlled alongside design documentation in the project repository and remain internal to the team unless explicitly shared for external review.
Changes to checklist scope or content follow a documented change-control process, with major changes triggering a version increment and minor editorial changes noted in a changelog.

Automation is encouraged: repository or CI/CD pipelines can enforce the presence of an up-to-date checklist, verify that all items are answered, and warn when “No” responses lack associated tickets or follow-up actions.

5. Best Practices, Patterns & Reference Architectures

This section introduces a living catalog of Secure-by-Design examples you can study and reuse. Each example is opinionated but minimal. It illustrates how the Section 3 principles appear in real designs (trust boundaries, authN/Z, data protection, resilience, observability) without product-specific noise. Treat these as starting points to adapt to your context. The catalog is community-maintained and will grow as new cases are contributed. The catalog is categorised as follows:

- Reference Architectures: End-to-end sample systems that demonstrate multiple SbD principles working together.
- Patterns Library: Reusable building blocks you can drop into many designs (e.g., API gateways, event-driven messaging, zero-trust service isolation).

These examples demonstrate how the framework's principles (Section 3) are applied in real designs, so you can adapt them with confidence. They are opinionated but minimal, showing exactly how key controls—such as trust boundaries, authentication/authorization, data protection, resilience, and observability—are implemented, without overwhelming you with product-specific details.

The catalog is community-maintained and will grow as new cases are contributed. Rather than listing samples here, the most up-to-date collection is always available in the repository:

View the SbD Catalog of Best Practices, Patterns & Reference Architectures

5.1 Contribution

We welcome community contributions to the Secure-by-Design Best Practices, Patterns & Reference Architectures. If you would like to add a new sample pattern or reference case, please review the Contributing Guidelines.

If you are contributing specifically to the Catalog, please also ensure your submission considers the following additional points:

- Each case should include a README with overview, diagram, controls spotlight, design notes/ADRs, sample SbD review checklist, and operational notes.
- Trust boundaries must be explicit, with clear enforcement points.
- Secure defaults should be applied throughout (TLS, least privilege, validation at first hop).
- Include a completed SbD review checklist (see Section 4 for reference).

5.2 Disclaimer

Samples are **educational**. Always validate against your project's security requirements, threat model, and regulatory context. Adapt configs and policies before deploying to production.

6. Contribution Guide & Project Governance

The **OWASP Secure-by-Design Framework** is an open, community-driven project. Contributions are encouraged from security engineers, product teams, architects, and anyone passionate about advancing secure design practices. Community input is welcome across all parts of the framework—including the process, principles, checklist, and the Catalog of Best Practices, Patterns, and Reference Architectures.

Please review the Contributing Guideline in the repository for details on how to contribute. If you plan to add a new sample pattern or reference case, also review the contribution guidance in the Catalog (Section 5.3) to ensure consistency and quality.

7. Roadmap and Future Work

The OWASP Secure-by-Design Framework progresses in phases to ensure quality, community feedback, and adoption.

- **Phase 1: Initial Draft (Q2 2025)**
  ✅ *Completed.* Defined the core Secure-by-Design principles, structured the framework into modules, and published the initial draft for feedback.

- **Phase 2: Community Contributions (Q3 2025 – Q1 2026)**
  Current phase. The project is now open for community input, real-world case studies, and best practices. OWASP meetups and discussions will help refine the guidance and grow the Catalog of patterns and reference architectures.

- **Phase 3: Formal OWASP Framework Release (Q2 – Q4 2026)**
  Publish Version 1.0 of the Secure-by-Design Framework, align with other OWASP projects (SAMM, ASVS, Cheat Sheets), and provide training/workshops.

- **Phase 4: Continuous Improvement (ongoing from Q4 2026 onward)**
  Adapt the framework to new threats, evolving technologies, and community needs. Collaborate with related OWASP projects to ensure alignment and maintain relevance.

8. Licensing and Usage Rights

The OWASP Secure-by-Design Framework is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0).

This means you are free to:

- **Share** — copy and redistribute the material in any medium or format.
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Full license text: https://creativecommons.org/licenses/by-sa/4.0/