OWASP Secure Coding Practices Checklist

An OWASP secure coding practices checklist breaks the full OWASP secure coding practices framework into pass/fail items organized by software development lifecycle phase predevelopment, design, implementation, testing, deployment and maintenance so a team can apply it directly during code review rather than treating it as a document read once and filed away. This guide is built as a working reference: skim to the phase you're in, run down the items and check them off.

Why Use a Secure Coding Checklist Instead of Reading the Full Guide

OWASP's Secure Coding Practices guide is comprehensive, but comprehensive and usable during a fifteen minute pull request review are two different things. A checklist strips the explanation down to actionable yes/no items, organized so a reviewer can quickly identify which items apply to the specific change in front of them rather than mentally scanning fourteen practice areas every time.

The most effective way to use a checklist like this is to map sections to the type of change being reviewed: a pull request adding a login flow triggers the authentication, session management and access control sections; a pull request adding file upload functionality triggers the file handling and input validation sections. Treating it this way turns a general purpose reference into something that actually gets used on every relevant change.

It also helps to remember that a checklist is a tool within a larger discipline, not a substitute for it. Secure coding is one piece of web application security as a whole, alongside infrastructure hardening, network controls and ongoing monitoring a perfectly checked pull request still ships inside an application that needs the rest of that discipline applied around it. The checklist below is deliberately scoped to what a developer and reviewer can actually control at the code level, phase by phase.

PreDevelopment Checklist

Before a single line of feature code gets written, a few structural decisions determine how much security work happens by default versus how much has to be bolted on later.

- Security requirements are defined alongside functional requirements, not added afterward.
- Data the feature will handle is classified by sensitivity (public, internal, confidential, regulated).
- Trust boundaries are identified at every point where data crosses from a lowertrust zone to a highertrust one.
- Existing libraries and frameworks are checked for known vulnerabilities before being added as dependencies.
- Abuse cases are written alongside use cases of how this feature could be misused, not just how it is meant to be used.

Design Phase Checklist

Designphase security is the cheapest security work a team ever does, since catching an architectural gap here costs a fraction of catching the same gap in production.

- Threat modeling has been performed on the proposed architecture (STRIDE, attack trees, or an equivalent lightweight routine).
- Authentication and authorization models are defined explicitly, not left implicit in individual endpoints.
- The principle of least privilege is applied to every new service account, database role and API scope.
- Default configurations are secure by design; insecure options require explicit action to enable, not to disable.
- Encryption requirements (in transit and at rest) are specified for any sensitive data the design touches.

Implementation Phase Checklist

This is where most of OWASP's checklist items concentrate, since implementation is where the majority of codinglevel vulnerabilities actually get introduced.

Input Validation

- All input from an untrusted source is validated server side.
- Validation uses allowlists (define what's valid) rather than blocklists (try to exclude what's invalid).
- A single, centralized validation routine is used rather than scattered ad hoc checks.
- Invalid input is rejected outright, not silently "corrected" or sanitized.

Output Encoding

- Output is encoded for the specific context it's rendered into HTML body, HTML attribute, JavaScript, CSS, or URL.
- A well tested encoding library is used rather than custom encoding logic.
- This applies to the XSS labs, which walk through exactly this class of gaps in deliberately vulnerable code.

Authentication and Session Management

- Passwords are hashed with Argon2id or an accepted fallback (scrypt, bcrypt) , never a fast, unsalted hash.
- Multifactor authentication is available and enabled by default for sensitive accounts.
- Login and password reset flows don't reveal whether a given account exists.
- Session tokens are generated with a cryptographically secure random number generator and sufficient entropy.
- Session cookies set `HttpOnly`, `Secure` and an appropriate `SameSite` value.
- The session ID is regenerated immediately after login and any privilege change.
- Sessions are invalidated serverside on logout, not just cleared clientside.

Access Control

- Authorization checks happen server side on every request, not just once at login.
- Access control defaults to deny, with explicit grants rather than explicit restrictions.
- Objectlevel permissions are verified if a user can't access another user's data by changing an ID.
- These checks are the exact category exercised in our source code review practice, which traces an authorization check from request to data.

Cryptography

- No custom built encryption, hashing, or random number generation anywhere in the codebase.
- Keys are never hardcoded, never reused across unrelated purposes and stored in a vault or key management system.

- TLS is enforced for all data in transit; sensitive data at rest is encrypted.

Error Handling and Logging

- Users receive generic error messages; full detail (stack traces, internals) is logged server side only.
- Failure defaults to a denied or locked state, never an open one.
- Security relevant events (auth attempts, access control failures, admin actions) are logged with enough context to reconstruct what happened.
- Logged data derived from user input is encoded and sensitive fields (passwords, tokens, full card numbers) are masked or truncated.

Secrets Management

- No API keys, passwords, or credentials are hardcoded anywhere in source code.
- Secrets are loaded from environment variables or a dedicated secrets vault.
- Every commit is scanned for accidental secret exposure before merge.

Testing Phase Checklist

- Static analysis (SAST) runs automatically on every commit in the CI/CD pipeline.
- Dynamic analysis (DAST) runs against deployed builds in staging.
- Dependency scanning flags known CVEs in third party libraries before release.
- Manual, security focused code review is applied to high risk components authentication, payment processing, access control.
- Automated code review findings are treated as a first pass requiring human verification, not a final verdict.
- AIassisted or AIgenerated code changes go through the same review, given that a meaningful share of AI output carries known OWASP Top 10 flaws see our guide on AI code review for how tooling adapts to this specifically.

Deployment Checklist

- Default accounts and credentials are removed or rotated before going live.
- Unnecessary services and ports are disabled.
- TLS is enforced, with modern protocol versions and appropriate certificate validation.
- Security headers are set (CSP, HSTS, XContentTypeOptions and equivalents for the platform).
- Secrets are confirmed to be sourced from a vault, not from committed configuration files or environment files bundled into the deployment artifact.
- Debug mode and verbose error output are disabled in production.

Maintenance Checklist

Security work doesn't end at release; it is the phase most checklists skip and the one where quiet drift accumulates fastest.

- Security advisories are monitored for every dependency in active use.
- A defined patch timeline is enforced for known vulnerabilities, prioritized by severity.
- Security logs are reviewed on a regular cadence, not only after an incident is already suspected.
- Periodic penetration testing and code audits are scheduled, not left to happen only after a scare.
- Deprecated or unused code paths and dependencies are removed rather than left dormant.

LanguageSpecific Checklist Notes Java, Python and Node.js

The implementation phase items above apply regardless of language, but a few checklist items deserve extra scrutiny depending on your stack, since each language's runtime quietly reintroduces certain risks in its own way.

Java codebases need extra attention on deserialization and XML parsing, specifically native object deserialization of untrusted data and XML parsers with external entity processing left enabled are two of the most common ways a Java application fails several implementation phase items at once. `PreparedStatement` with bound parameters should be the default for every query, not an exception reserved for risky ones. Our Java security code review guide walks through exactly these patterns with real vulnerable code examples, which pairs well with running this checklist against an actual Java pull request.

Python codebases should add an explicit check for `eval()`, `exec()` and `pickle.loads()` on untrusted data, none of which appear as a distinct item above but all of which violate the input validation and cryptography sections in a Python-specific way. Django and Flask's builtin ORM parameterization and template autoescaping cover most of the input validation and output encoding items automatically; the checklist failure usually happens when a developer bypasses those framework defaults with raw SQL or an unescaped render.

Node.js codebases should verify that user input never reaches a template engine, `child_process` call, or database query unescaped and that dependency scanning (a testingphase item above) runs against `npm` packages specifically, given how quickly transitive dependencies can introduce a known vulnerable package into a project without a developer directly adding it.

Common Checklist Mistakes Teams Make

A checklist only works if it's actually used consistently and a few patterns undermine that in practice more often than any single missing item does.

Treating it as a onetime gate instead of a recurring review. Running through the checklist once before initial launch and never again leaves every subsequent feature unchecked. The checklist needs to apply to every meaningful pull request, not just the original release.

Applying every item to every change regardless of relevance. A checklist that forces a developer to confirm cryptography items on a change that touches none is friction without value

and teams that do this tend to start rubber stamping the whole list. Tagging each pull request with only the relevant sections keeps the checklist fast enough to actually survive contact with a real sprint.

No ownership for keeping it current. A checklist copied once from a general OWASP reference and never updated drifts out of sync with the team's actual stack. Someone, usually a security champion or lead engineer, needs to update it as the stack and the threat landscape evolve.

Skipping the maintenance phase items entirely. Most teams are diligent about prerelease checks and far less consistent about ongoing patch monitoring and log review, which is exactly the phase where slowmoving vulnerabilities accumulate unnoticed.

Treating a passed checklist as proof of security rather than a floor. Completing every item on this list substantially reduces risk, but it does not guarantee an application is free of vulnerabilities, business logic flaws, novel attack patterns and mistakes outside the checklist's scope can still slip through. The checklist is a baseline that catches the wellunderstood, recurring categories of mistake; it is not a substitute for periodic penetration testing or a fresh set of eyes reviewing the application as a whole and treating it as the final word tends to create exactly the false confidence it's meant to prevent.

What Skipping This Checklist Actually Costs

It is worth being concrete about why each item above earns its place on the list. Every section maps to one of the web app security threats that account for the large majority of realworld breaches: injection, broken authentication, broken access control and cryptographic failures are not hypothetical categories, they are the specific, recurring failure modes this checklist is built to intercept before code ships. A checklist item that feels like friction during a busy sprint is, in nearly every case, standing in for a vulnerability class that has already caused real damage somewhere else. Treating the checklist as optional under deadline pressure is, in effect, deciding that this specific sprint's code doesn't need the protection every other sprint's code needs.

Turning This Checklist Into Practiced Habit

A checklist tells you what to check; it does not build the instinct to recognize a violation on sight. That instinct comes from handson practice against real vulnerable code. Our challenges index covers the vulnerability classes that checklist references, from injection through access control, while our web application hacking labs let you exploit these gaps from the attacker's perspective before fixing them, a combination that tends to make checklist items stick far better than reading them in isolation.

Conclusion

A checklist is only as useful as the workflow it's embedded in. Organized by SDLC phase and mapped to specific pull request types, the items above turn OWASP's broad guidance into something a team can actually run consistently, rather than a reference document everyone agrees is important and nobody actually opens. For the complete narrative explanation behind every item here the reasoning, the real world statistics and the full 14area breakdown see our comprehensive OWASP secure coding practices guide.If you are new to the platform, start from appsecmaster.net for a guided entry point.

Frequently Asked Questions (FAQs)

Should this checklist be run on every pull request, or only before release?

Every meaningful pull request that touches the relevant category. Reserving checklist review for prerelease gates means months of code-ship unchecked and finding a vulnerability that late is far more expensive than catching it during the original review.

How do we avoid the checklist becoming a rubberstamp exercise?

Tag each pull request with only the sections relevant to what it changes, rather than requiring every item on every change. A shorter, targeted checklist gets followed consistently; a long, generic one tends to get skimmed and approved without real scrutiny.

Can automated tools cover this whole checklist on their own?

No. Static analysis and dependency scanning cover a meaningful share of the implementation phase items: hardcoded secrets, known/vulnerable patterns, some encoding checks but access control logic and business logic dependent items still require a human reviewer who understands what the feature is supposed to restrict.

What is the most commonly skipped section of this checklist?

Maintenance. Prerelease and implementation items get consistent attention because they're tied to a specific release deadline; ongoing dependency monitoring, patch timelines and periodic log review have no natural deadline forcing them to happen, so they're the section most likely to drift.

How is this checklist different from the OWASP Top 10?

The OWASP Top 10 ranks the most critical and prevalent risk categories; this checklist turns OWASP's Secure Coding Practices guidance into specific, checkable items mapped to each phase of the development lifecycle. The Top 10 tells you what to prioritize; this checklist tells you what to verify.