OWASP Security for Entry Developers & Vibe Coders

A practical, plain-language guide to the **OWASP Top 10 (2025)**, the ten most critical web application security risks, written specifically for junior developers and **vibe coders** shipping AI-generated apps as real products.

When you build with Claude, ChatGPT, Cursor, Copilot, or any AI coding assistant, the code *looks* professional. It compiles, the tests pass, the demo works. But AI models are trained to produce code that **functions**, not code that is **secure**. They will happily write an authentication system that leaks tokens, a database query that allows injection, or a file upload endpoint that lets attackers run arbitrary code and present it confidently.

SENTIDO
SOFTWARE MADE SIMPLE

What is OWASP?

The **Open Worldwide Application Security Project (OWASP)** is a non-profit foundation that has, since 2001, become the de-facto standard for web application security awareness.

Their flagship deliverable is the **OWASP Top 10**, a ranked list of the most critical security risks facing web applications today.

The list is **data-driven**. The 2025 edition was built from analysis of over **175,000 Common Vulnerability and Exposure (CVE) records** and feedback from security practitioners across the globe, covering data from more than **2.8 million applications**. It is updated roughly every **four years**.

Why It Matters

- **Industry baseline** - Security audits, penetration tests, and compliance frameworks (PCI-DSS, ISO 27001, SOC 2) all reference the **OWASP Top 10**
- **Insurance & liability** - Cyber insurers expect OWASP awareness; gaps may void a pay-out
- **Customer trust** - B2B clients increasingly require OWASP attestation before signing
- **Free and open** - The full standard is at owasp.org/Top10

Cheat Sheet Series

One-page guides per security topic

OWASP ZAP

Free vulnerability scanner

OWASP ASVS

Detailed verification standard for serious products

Dependency-Check

Free supply chain scanner

Why Vibe Coders Are at Higher Risk

Vibe coding, describing a feature in natural language and reviewing whatever the AI produces, accelerates delivery dramatically. It also concentrates risk in ways traditional development does not.

AI models are trained on **public code**, which includes a vast amount of **insecure code**: tutorial blogs, Stack Overflow answers, abandoned GitHub repos. When you ask for "a login form," the model averages across what it has seen.

> Average internet code is not secure code.

1. ### Confidence Without Comprehension

   The code works in the happy path, so it ships. The attacker's path is never tested.

2. ### Copy-Paste Credentials

   AI examples often hardcode API keys "for clarity." Vibe coders accidentally leave them there.

3. ### Outdated Patterns

   Models suggest deprecated, vulnerable libraries or insecure defaults e.g. MD5 for passwords, `eval()` for parsing.

4. ### Slopsquatting

   Hallucinated package names have been exploited by attackers registering malicious packages under names AI commonly invents.

5. ### No Threat Modelling

   Senior developers ask "what could go wrong?" Vibe coders ask "does it run?"

⚠ Assume that anything an AI generates touching authentication, user input, file handling, payments, or external services is **insecure until proven otherwise**.

SENTIDO

SOFTWARE MADE SIMPLE

A01:2025 - Broken Access Control

⚠ RANKED #1 MOST SERIOUS RISK | FOUND IN 3.73% OF APPLICATIONS

Access control decides who can see or do what. It is *broken* when a user can perform actions or access data they shouldn't, for example:

- Viewing another user's invoices
- Deleting someone else's post
- Calling an admin-only endpoint by guessing the URL.

In 2025, **Server-Side Request Forgery (SSRF)** was rolled into this category. This is where an attacker tricks your server into making requests on their behalf, often to internal cloud metadata services.

Vibe Coding Failure Modes

- AI generates `/api/users/:id` endpoints that fetch by ID without checking the requester owns that ID
- Admin checks done in the frontend only, this is easily bypassed by calling the API directly
- File download endpoints that accept any path: `/files?name=../../etc/passwd`
- Webhook URLs accepted from users without validation, enabling SSRF into cloud metadata (`http://169.254.169.254/`)

How to Defend

- **Deny by default.** Every endpoint requires explicit permission, not the absence of denial
- Enforce ownership: `WHERE user_id = current_user.id` on every query touching user data
- Never trust client-side role checks, re-verify on the server
- Validate and allow list any URL your server will fetch

A02:2025 - Security Misconfiguration

SURGED FROM #5 TO #2

FOUND IN 3.00% OF APPLICATIONS

The software is fine, but the way it's configured is not. Default passwords left in place, debug endpoints exposed in production, overly permissive Cross Origin Resource Sharing (CORS), verbose error messages leaking stack traces, public storage buckets, missing security headers.

> Misconfiguration is the silent killer of otherwise well-written applications.

Permissive CORS

AI can suggest `cors({ origin: '*' })` to "make it work", allowing everything and it never gets tightened.

Committed Secrets

`.env` files committed to public GitHub repos, exposing credentials to anyone.

Debug Mode in Production

`DEBUG = True` left on in Django/Flask, exposing secrets and stack traces on errors.

Open Cloud Defaults

Supabase, Firebase, and S3 Storage resources deployed with default open policies.

How to Defend

- Maintain a hardening checklist per environment (dev / staging / prod)
- Disable verbose errors in production. Log server-side and show users a generic message
- Set security headers: `Content-Security-Policy`, `Strict-Transport-Security`, `X-Frame-Options`
- Scan for committed secrets with **gitleaks** or **trufflehog** before every push
- Review all default cloud permissions on day one

SENTIDO

SOFTWARE MADE SIMPLE

A03:2025 - Supply Chain Failures

NEW EXPANDED CATEGORY | HIGHEST AVERAGE EXPLOIT & IMPACT SCORES

You did not write 95% of the code in your application, you imported it.

Every `npm install`, `pip install`, or `composer require` pulls in dependencies, and their dependencies, and their dependencies.

A compromise anywhere in that chain becomes your compromise. This category covers vulnerable libraries, malicious packages, compromised build pipelines, and tampered distribution.

Vibe Coding Failure Scenarios

- **Slopsquatting** - AI hallucinates a package name; an attacker registers it with a malicious payload
- Blindly accepting AI suggestions for obscure packages without checking download counts or maintainer history
- Outdated dependencies with public CVEs because nobody runs `npm audit`
- `curl | bash` install commands suggested by AI without inspection

How to Defend

- **Verify every package** the AI suggests, check npm/PyPI page, download counts, last publish date
- Pin versions and use lock files (`package-lock.json`, `poetry.lock`)
- Run dependency scanners weekly: `npm audit`, `pip-audit`, GitHub Dependabot, Snyk
- Use **Subresource Integrity (SRI)** hashes for frontend CDN imports
- Treat your CI/CD pipeline as production, its credentials own everything

A04:2025 - Cryptographic Failures

FOUND IN 3.80% OF APPLICATIONS

OFTEN LEADS TO SENSITIVE DATA EXPOSURE

Failures in how data is protected: at rest, in transit, or in storage. This includes using weak algorithms, hardcoded keys, transmitting passwords in clear text, or storing sensitive data unencrypted. AI models frequently produce "simple examples" that use dangerously outdated cryptographic approaches.

Weak Password Hashing

Storing passwords with MD5, SHA-1, or even plaintext because the AI showed a "simple example."

Hardcoded Secrets

API keys, JWT secrets, or encryption keys embedded directly in source files.

HTTP Instead of HTTPS

Unencrypted endpoints exposed, including for "internal" services that aren't truly internal.

Weak JWT Signing

Tokens signed with the `none` algorithm or weak secrets like `"secret"`.

How to Defend

- Passwords: use functions like **bcrypt**, **argon2**, or **scrypt** - never MD5 or SHA-256 alone
- Secrets: store in environment variables or secret managers (AWS Secrets Manager, Azure Key Vault, HashiCorp Vault), never in code
- Force HTTPS everywhere with HSTS, no exceptions for "internal" services
- Use vetted libraries (libsodium, platform crypto). **Never write your own crypto**
- Rotate keys on schedule, and immediately if there's a suspected leak

> Never Write Your Own Crypto!

A05:2025 - Injection

SQL INJECTION | COMMAND INJECTION | CROSS-SITE SCRIPTING (XSS)

User input gets interpreted as code or commands. An attacker types something into a form and your application runs it as a database query, shell command, or browser script. This is one of the oldest and most exploited vulnerability classes, and AI-generated code reproduces it constantly.

Vibe Coding Failure Modes

- AI generates SQL by string concatenation e.g. a user types `' OR 1=1 --` and then writes a command to expose or delete the database
- User input rendered into HTML without escaping, allowing `<script>` tags to execute in other users' browsers
- `exec()` or `os.system()` calls that include user-controlled strings
- **LLM prompt injection** - user input flowing into prompts your own AI agents execute

How to Defend

- **Always use parameterised queries / prepared statements** - never send plaintext into SQL commands
- Escape output based on context (HTML, JS, URL, and SQL each have different rules)
- Validate input against an **allow list** (what is permitted), not a deny list
- For shell commands, prefer language APIs - never pass user input to a shell
- Treat LLM output as untrusted input if it flows into other systems

---

A06:2025 - Insecure Design

The code is implemented correctly, but the design itself is flawed. Examples:

- A password reset that uses "What's your mother's maiden name?" as the only barrier
- A shopping cart that lets discount codes stack infinitely
- An API rate limit that resets per IP when users are behind shared NATs.

This is a class of bugs no automated scanner will find.

Business Logic Abuse

AI generates working features without considering abuse: voucher stacking, free trial farming via email aliasing, race conditions in balance updates.

No Abuse Modelling

Only "how will this be used?" is considered, never "how will this be abused?"

Logical Auth Bypasses

Authentication flows that look secure but contain logical bypasses, e.g. password reset emails sent to user-supplied addresses.

How to Defend

- Before building anything sensitive, ask: **"How would a malicious user abuse this?"** Write that down and address each scenario
- Threat-model for your critical flows: payments, authentication, file uploads, anything touching money or **Personally Identifiable Information** (PII)
- Apply rate limits, transaction limits, and anomaly detection on financial paths
- Have a second human review the design of any feature touching payments, auth, or admin actions

---

SENTIDO

SOFTWARE MADE SIMPLE

A07:2025 - Authentication Failures

Weaknesses in how users prove who they are:

- Weak password policies
- Missing Multi-Factor Authentication (MFA)
- Poor session management
- Credential stuffing vulnerability
- Account enumeration through error messages.

> Authentication is the front door of your application

AI-generated auth code is frequently the weakest part of any vibe-coded product.

Vibe Coding Failure Modes

- Hand-rolled login systems instead of using battle-tested libraries
- Sessions that never expire, or tokens stored in `localStorage` where any XSS reads them
- "User not found" vs "Wrong password" error messages - letting attackers enumerate valid accounts
- No rate limiting on login, enabling credential stuffing at scale
- Password reset tokens that never expire or aren't single-use

How to Defend

- **Don't build your own auth.** Use Auth0, Clerk, Supabase Auth, AWS Cognito, NextAuth, or your framework's vetted module
- Enforce MFA for any privileged or sensitive action
- Generic error messages: "Invalid credentials," not "User not found"
- Rate-limit login attempts per account and per IP
- Set short session timeouts; invalidate on logout server-side, not just client-side

---

A08 & A09:2025 - Integrity Failures & Logging Gaps

A08 - Software or Data Integrity Failures

- Trusting code, updates, or data without verifying their integrity.
- Auto-updating from an untrusted source, deserialising untrusted data, accepting deployment artefacts without signing.
- Deserialising user-controlled JSON/pickle/YAML into objects that can execute code
- Loading scripts from a Content Delivery Network with no integrity hash
- Trusting webhook payloads without verifying the signature
- Auto-deploying from a Git push without code review

> ⓘ Always verify webhook signatures - Stripe, GitHub, and Slack all provide request headers for this. Never deserialise untrusted data into executable types.

A09 - Security Logging & Alerting Failures

If you can't see attacks happening, you can't stop them. New emphasis in 2025: having logs but no **alerting** on them. Logs nobody looks at are theatre.

- No logging on auth events, admin actions, or payment flows
- Logging sensitive data (passwords, tokens) to log files that are themselves a breach risk
- No alerts when something abnormal happens means that 10,000 failed logins overnight goes unnoticed

> ⓘ Centralise logs using software such as New Relic, Datadog, Logtail, CloudWatch, Sentry). Alert on anomalies: spike in 5xx errors, repeated failed logins, new admin accounts, unusual outbound traffic.

A10:2025 - Mishandling of Exceptional Conditions

NEW FOR 2025

How your application behaves when something unexpected happens, For example:

- Catching exceptions and silently ignoring them.
- "Failing open" instead of failing closed i.e. an auth check that errors out and *grants* access.
- Logical errors when inputs are at the edges of what was anticipated.

This category is particularly dangerous because the failures are invisible in normal testing.

Swallowed Exceptions

AI generates `try { ... } catch (e) { /* ignore */ }` blocks that silently discard security-relevant errors.

Fail-Open Auth

Auth middleware that returns `next()` if the token check throws an error, granting access on failure.

Race Conditions

Concurrent balance or inventory updates the AI didn't consider, enabling double-spend or oversell exploits.

Unhandled Edge Cases

Empty arrays, null values, integer overflow, or very large inputs crashing the app or producing unexpected behaviour.

Fail closed, not open. If a security check errors out, deny access. Never silently swallow exceptions - log them and decide deliberately. Use database transactions and locking on operations that change critical state (money, inventory, permissions).

SENTIDO
SOFTWARE MADE SIMPLE

Pre-Release Security Checklist

Before charging users for anything you built with AI assistance, verify these items. None require deep security expertise. If you cannot tick every box, your product is not ready to charge for.

Fix the Gaps First.

🔑 Secrets & Configuration

- [ ] No API keys, passwords, or tokens in source code or Git history
- [ ] All secrets in environment variables or a secret manager
- [ ] `.env` and similar files in `.gitignore`
- [ ] Debug mode disabled in production
- [ ] Verbose error messages disabled for end users

📦 Dependencies

- [ ] `npm audit` / `pip-audit` clean, or all findings triaged
- [ ] Every AI-suggested package manually verified as reputable
- [ ] Lock files committed
- [ ] Dependabot or equivalent enabled

🔐 Authentication & Access Control

- [ ] Using a vetted auth provider (not hand-rolled)
- [ ] MFA available for privileged accounts
- [ ] Every API endpoint checks permissions server-side
- [ ] Session timeouts set; rate limiting on login and password reset

🛡️ Data & Input

- [ ] HTTPS enforced everywhere (no HTTP fallback)
- [ ] Passwords hashed with bcrypt, argon2, or scrypt
- [ ] All database queries parameterised
- [ ] File uploads restricted by type, size, and stored outside web root

📊 Operational

- [ ] Centralised logging in place
- [ ] Alerts configured for failed logins, errors, and unusual traffic
- [ ] Incident response plan written (even one page)
- [ ] Backup and restore tested at least once

AI-Driven OWASP Audit Prompt

Below is an execution-ready prompt you can paste into any AI coding assistant with codebase access. It instructs the AI to audit your code against the OWASP Top 10, generate proof-of-vulnerability tests, and propose surgical fixes, with your approval at every step. The AI **must have access to your full repository** for this to work, not just a single file pasted into chat.

Claude Code (Recommended)

Save as `.claude/commands/owasp-audit.md` in your repo root. Invoke with `/owasp-audit`, or paste into a Claude Code chat opened at the repo root.

Cursor

Open the repo. In chat, type `@codebase` then paste the prompt.

GitHub Copilot Chat

Open the workspace in VS Code. In chat, type `@workspace` then paste the prompt.

ChatGPT Codex / Aider

Open a session in the repo root and paste the prompt directly. For large repos with Aider, pre-load critical files with `/add`.

Pre-Audit Setup

- Commit your repo before running, the audit will propose changes and you need a baseline to diff against
- Create a dedicated branch: `git checkout -b security/owasp-audit`
- Ensure dependency lock files are committed and your test framework runs cleanly
- Create a `/security/` folder in the repo root, the AI will write its report there

---

SENTIDO

SOFTWARE MADE SIMPLE

The Audit Prompt

Copy and paste the following prompt into your chosen AI coding assistant. This prompt drives a structured, seven-phase audit from reconnaissance through to verified fixes, all with your explicit approval before any code is changed.

You are an expert Application Security Auditor specialising in the OWASP Top 10 (2025). Perform a comprehensive security audit of the codebase and produce an actionable, prioritised remediation plan backed by proof-of-vulnerability tests.

CORE PRINCIPLES
- Use evidence from the actual code, never assumptions
- Every finding includes: file path, line number, exact vulnerable code, exploit scenario, recommended fix
- Never fabricate findings - if uncertain, mark "needs human review"
- Default recommendations to "fail closed" behaviour
- Prefer minimal, surgical fixes over rewrites

PHASE 1 - RECONNAISSANCE
Map the codebase. Output: stack summary, auth mechanism, database/ORM, external services, all user input entry points, secrets management approach, dependency manifests, trust boundaries diagram.

PHASE 2 - OWASP TOP 10 ANALYSIS
For each category A01–A10, audit relevant code paths. For every finding output: Category, Severity, File path and line numbers, Vulnerable code snippet, Concrete exploit scenario, Recommended fix with code, Confidence level.

PHASE 3 - TEST GENERATION
For every Critical and High finding, generate a failing security test demonstrating the vulnerability and the same test passing after the fix. Place tests in the existing test directory structure.

PHASE 4 - REMEDIATION PLAN
Prioritised action list: Immediate (Critical), This week (High), This sprint (Medium), Backlog (Low). Include estimated effort (S/M/L), files to modify, automatable yes/no.

PHASE 5 - HARDENING BEYOND THE TOP 10
Recommend defences the codebase lacks with implementation snippets specific to THIS stack: security headers, rate limiting, CSRF protection, secret scanning in CI, dependency scanning, centralised structured logging.

PHASE 6 - APPLY FIXES (REQUIRE CONFIRMATION)
Before modifying any file: show the unified diff, wait for explicit user confirmation, apply only confirmed changes, commit with messages referencing the OWASP category.

PHASE 7 - VERIFICATION
Run generated security tests and existing test suite. Output final report at /security/owasp-audit-report.md.

Begin with Phase 1 now and confirm the stack summary with me before proceeding to Phase 2.

Audit Prompt Variants & Iteration Cadence

Prompt Variants

Adjust the prompt by replacing the final instruction based on what you need:

1

Baseline Scan (Read-Only)

Run Phases 1 and 2 only. Do not generate tests or fixes. Output the report and stop.

2

Test Generation Only

Run Phases 1, 2, and 3. Do not propose fixes, only the proof-of-vulnerability tests.

3

Full Audit + Supervised Fixes

Default prompt above. Recommended for most projects.

4

Re-Audit After Fixes

A previous audit is at `/security/owasp-audit-report.md`. Re-run Phases 1, 2, and 7. Confirm resolved findings and identify regressions.

Iteration Cadence

The audit is not a one-time event. Re-run the prompt:

- After every significant feature merge into main
- Before every public release
- After every dependency update
- Quarterly at minimum, even if nothing has changed (new CVEs surface constantly)

Track findings over time in `/security/findings.md`. The goal is to drive Critical and High counts to zero and keep them there.

> ⚠ Treat the AI audit as the **first line of defence**, not the last.

What This Prompt Will NOT Do

Understanding the limits of an AI-driven audit is as important as running one. The prompt is a powerful first-line tool, but it has hard boundaries that every coder must understand before relying on it.

No Penetration Testing

It does not replace a professional penetration test for production applications handling payments, health data, or regulated information.

No Business Logic Coverage

It cannot catch every business logic flaw, some require human threat modelling and domain knowledge.

No Runtime Testing

It only performs static analysis of code, it does not test runtime behaviour against a live attacker.

No Infrastructure Audit

It cannot verify your hosting, network, or cloud configuration outside of what's expressed in code. Use AWS Inspector, GCP Security Command Center, or Azure Defender for those layers.

> Treat the AI audit as the **first line of defence**, not the last. For applications handling payments, health data, or significant user data, engage a qualified security professional.

Legal Disclaimer

The information in this document represents general advice and educational content. Security decisions should always be validated within your own environment. This document is not a substitute for a professional security audit, penetration test, or qualified security review.

Your Responsibility

You are responsible for verifying the security implications of any configuration or workflow adopted from this document.

Code Validation

You are responsible for validating all AI-generated code before it is deployed to any environment, production or otherwise.

Consequences

You are responsible for the consequences of implementation. Proceed with caution.

For applications handling payments, health data, regulated information, or significant user data, engage a qualified security professional. Sentido and its author accept no liability for damages or issues arising from the use of the information contained in this document.