

THE COMPLETE PRACTICAL GUIDE

Web Application Security for Developers

OWASP Top 10 in Practice

Access Control · Injection · XSS · Auth · SSRF ·
Supply Chain · DevSecOps · AI Feature Security

28 CHAPTERS · VULNERABLE vs. FIXED CODE

THENEXBOOKS

Web Application Security for Developers

OWASP Top 10 in Practice

Understand the Risks, Fix the Code, Build Security into Every Release

First Edition

THENEXBOOKS

Web Application Security for Developers
OWASP Top 10 in Practice

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Following this book does not guarantee that any application is free of vulnerabilities. Security guidance, standards, and libraries change; always consult current official documentation.

Responsible use. The attack examples in this book exist to help you understand and prevent vulnerabilities in software you build. Only test systems that you own or are explicitly authorized in writing to test. Unauthorized testing may be illegal.

OWASP and the OWASP Top 10 are trademarks of the OWASP Foundation. This book is not affiliated with or endorsed by the OWASP Foundation. Other product names are trademarks of their respective owners; their use in this book is for identification only and does not imply endorsement.

Lattice Learn, the running example in this book, is fictional. Any resemblance to real companies or products is coincidental.

Preface

Most security vulnerabilities are not exotic. They are ordinary code: a query built with string concatenation, an endpoint that checks whether you are logged in but not whether the record is yours, a password reset token that never expires, a dependency nobody has updated in two years. They are written by capable developers under deadline pressure — and they are found, sooner or later, by someone who is looking.

That is good news. If vulnerabilities come from everyday code, developers are the people best placed to prevent them. You do not need to become a penetration tester. You need to recognize the patterns that lead to weaknesses, understand why the standard defenses work, and build habits and tooling that catch mistakes before they ship.

This book is a practical guide to exactly that. It is organized around the **OWASP Top 10** — a widely used awareness document for web application security risks — and goes further into authentication, sessions, OAuth 2.0, API security, secure design, the software supply chain, security testing, and the new risks that come with AI-powered features. Every risk is shown in realistic code, first vulnerable and then fixed.

Who This Book Is For

- **Web and backend developers** who want to write secure code by default and pass security reviews with confidence.
- **Frontend developers** who need to understand XSS, CSP, CORS, cookies, and token storage.
- **Tech leads, architects, and DevOps engineers** responsible for secure design, reviews, and pipelines.

What You Will Learn

- How web applications are attacked, and how to threat model a feature before you build it.
- Every OWASP Top 10 (2021) category in depth, with vulnerable and fixed code in TypeScript, Python, and Java.
- Access control that holds up in multi-tenant systems, and authentication, sessions, JWTs, OAuth 2.0, and OpenID Connect done right.
- Injection, XSS, CSRF, file upload, deserialization, and XXE defenses, plus Content Security Policy and security headers.
- Cryptography, secrets management, secure design, SSRF prevention, and the OWASP API Security Top 10.
- Configuration hardening, dependency and supply-chain security, logging and detection, security testing, and DevSecOps.
- How to secure AI-powered features against prompt injection, data leakage, and excessive agency.

How This Book Is Organized

Part I — Security Foundations (Chapters 1–4) explains how attacks work, how to threat model, and how to use OWASP resources. **Part II — Access Control and Identity** (Chapters 5–9) covers the most common and damaging class of flaws, along with authentication, sessions, and OAuth. **Part III — Injection and Untrusted Input** (Chapters 10–14) handles everything that comes from outside your trust boundary. **Part IV — Data Protection and Secure Design** (Chapters 15–19) covers cryptography, secrets, design flaws, SSRF, and API security. **Part V — Configuration, Supply Chain, and Operations** (Chapters 20–25) moves from code to the systems around it. **Part VI — Case Studies and Emerging Risks** (Chapters 26–28) brings everything together. The appendices provide a cheat sheet, code templates, a pre-release checklist, and a glossary. Each chapter ends with a **Key Takeaways** box.

A Note on OWASP Editions

This book uses the **OWASP Top 10 2021** categories (A01–A10) as its organizing framework. OWASP periodically updates the list; check owasp.org for the latest edition and how its categories map to the 2021 list. The underlying weaknesses and defenses in this book do not depend on the ranking of any particular edition.

The Running Example: Lattice Learn

Throughout the book we secure **Lattice Learn**, a fictional online learning platform. Students enroll in courses, watch lessons, take quizzes, and upload assignments; instructors publish content and grade work; organizations sign in with SSO and manage their members; a public API, payment webhooks, an "import course from URL" feature, and an AI study assistant round out the attack surface.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, standards references, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can lead directly to a vulnerability. Code blocks are labeled **Vulnerable** or **Fixed** — never deploy code labeled Vulnerable.

Contents

Preface

PART I — SECURITY FOUNDATIONS

1 Why Application Security Is a Developer's Job

2 How Web Applications Are Attacked: HTTP, Browsers, and Trust Boundaries

3 Threat Modeling for Developers

4 Using the OWASP Top 10, ASVS, CWE, and the API Security Top 10

PART II — ACCESS CONTROL AND IDENTITY

5 A01 Broken Access Control, Part 1: IDOR and Missing Function-Level Authorization

6 A01 Broken Access Control, Part 2: Multi-Tenancy, Path Traversal, and CORS

7 A07 Authentication Failures: Passwords, MFA, and Account Recovery

8 Sessions, Cookies, and Tokens: Secure Sessions and JWT Pitfalls

9 OAuth 2.0 and OpenID Connect Done Right

PART III — INJECTION AND UNTRUSTED INPUT

10 A03 Injection, Part 1: SQL and NoSQL Injection

11 A03 Injection, Part 2: OS Command, Template, and Header Injection

12 Cross-Site Scripting (XSS) and Content Security Policy

13 CSRF, Clickjacking, and Browser Security Headers

14 File Uploads, Insecure Deserialization, and XXE

PART IV — DATA PROTECTION AND SECURE DESIGN

15 A02 Cryptographic Failures: TLS, Password Hashing, and Encryption at Rest

16 Secrets Management and Key Handling

17 A04 Insecure Design: Business Logic Flaws and Secure Design Patterns

18 A10 Server-Side Request Forgery (SSRF)

19 API Security: The OWASP API Security Top 10 in Practice

PART V — CONFIGURATION, SUPPLY CHAIN, AND OPERATIONS

20 A05 Security Misconfiguration: Defaults, Errors, Cloud, and Containers

21 A06 Vulnerable and Outdated Components: Dependencies, SCA, and SBOMs

22 A08 Software and Data Integrity Failures: CI/CD Security and Signing

23 A09 Security Logging and Monitoring Failures: Detection and Response

24 Security Testing: SAST, DAST, Dependency Scanning, and Code Review

25 Secure SDLC and DevSecOps: Building Security into Every Stage

PART VI — CASE STUDIES AND EMERGING RISKS

26 Case Study: A Full Security Review of Lattice Learn

27 Case Study: Hardening the Lattice Learn Public API

28 Securing AI-Powered Features: Prompt Injection, Data Leakage, and Excessive Agency

APPENDICES

A OWASP Top 10 Cheat Sheet

B Secure Code Templates and Reference

C Pre-Release Security Checklist

D Glossary

PART I

Security Foundations

How attacks on web applications actually work, how to think like an attacker when you design a feature, and how to use OWASP's resources as a practical guide rather than a checklist to fear.

1 · Why Application Security Is a Developer's Job

2 · How Web Applications Are Attacked

3 · Threat Modeling for Developers

4 · Using OWASP Resources

CHAPTER 1 · PART I — SECURITY FOUNDATIONS

Why Application Security Is a Developer's Job

Most security weaknesses in modern web applications are written, reviewed, and shipped by the same people who build features. This chapter explains why that makes security part of your everyday engineering work, introduces Lattice Learn, the application we will secure throughout the book, and sets up the mindset, principles, and lab you will use in every chapter that follows.

A Bug That Nobody Called a Security Bug

Imagine a quiet Tuesday on the Lattice Learn engineering team. A developer is finishing a small feature: students should be able to see their quiz grades on a new dashboard. The React front end already knows the logged-in student's ID, so the developer adds an API route that takes a student ID in the URL and returns that student's grades. The tests pass, the reviewer approves the pull request because the code is tidy, and the feature ships the same afternoon.

Two weeks later, a curious student opens the browser's developer tools, notices the number in the request URL, and changes it. The API happily returns another student's grades. Then another. Nothing crashed, no alarm fired, and no firewall rule was broken. The system did exactly what the code told it to do.

This story contains no exotic hacking technique. There is no memory corruption, no zero-day, no clever cryptanalysis. There is a missing check, written by a competent developer who was thinking about the feature rather than about who might misuse it. That is the most common shape of a real-world application vulnerability, and it is the reason this book exists. The flaw above has a name — an insecure direct object reference, part of **A01 Broken Access Control** in the OWASP Top 10 2021 — and you will study it in depth in Chapter 5. For now, notice where it came from: application code, written by a developer, reviewed by a developer, and fixable only by a developer.

Vulnerable — src/routes/grades.ts

```
import { Router } from "express";
import { db } from "../db";
import { requireLogin } from "../middleware/auth";

export const gradesRouter = Router();

// Returns every quiz grade for the student in the URL.
gradesRouter.get("/students/:studentId/grades", requireLogin, async (req, res) => {
  const { rows } = await db.query(
    "SELECT quiz_id, score, graded_at FROM grades WHERE student_id = $1",
```

```
    [req.params.studentId]
  );
  res.json(rows);
});
```

The query is parameterized, so it is not vulnerable to SQL injection, and the route requires a login, so anonymous visitors are rejected. Many reviewers would stop there. What the code never asks is the question that matters: *is the logged-in user allowed to see this particular student's grades?* Authentication (who are you?) is present; authorization (may you do this?) is missing.

Fixed — src/routes/grades.ts

```
import { Router } from "express";
import { db } from "../db";
import { requireLogin } from "../middleware/auth";
import { canViewStudentGrades } from "../authz/policies";

export const gradesRouter = Router();

gradesRouter.get("/students/:studentId/grades", requireLogin, async (req, res) => {
  const viewer = req.user!; // set by requireLogin
  const studentId = req.params.studentId;

  // Deny by default: a student sees only their own grades; instructors and
  // org admins are checked against enrollment and tenant in the policy.
  if (!(await canViewStudentGrades(viewer, studentId))) {
    return res.status(404).json({ error: "not_found" });
  }

  const { rows } = await db.query(
    "SELECT quiz_id, score, graded_at FROM grades WHERE student_id = $1",
    [studentId]
  );
  res.json(rows);
});
```

The fix is a few lines long. It moves the decision into a named policy function that can be tested on its own, denies by default, and returns `404` rather than `403` so the response does not confirm which student IDs exist. The hard part was never the code; it was noticing that the question needed to be asked. Building the habit of asking such questions, systematically and early, is what this book is about.

What Application Security Actually Covers

Security is a broad field. Network engineers worry about firewalls and segmentation, infrastructure teams patch operating systems, and security operations teams watch alerts around the clock. **Application security** is the part of the field concerned with the software you build: its design, its code, its dependencies, its configuration, and the way it is built and deployed. If a weakness can be fixed by changing your application's source code, its build pipeline, or its runtime configuration, it is squarely in application security territory.

A useful way to describe what we are protecting is the classic trio of properties often called the CIA triad:

- **Confidentiality** — information is disclosed only to those allowed to see it. Another student's grades, an organization's invoices, and a user's password hash must stay private.
- **Integrity** — information and behavior change only in authorized ways. A student must not be able to raise their own quiz score or mark an unpaid subscription as paid.
- **Availability** — the system remains usable for legitimate users. An attacker should not be able to exhaust the video transcoding queue or lock every instructor out of their account.

Web applications add a few more properties that are worth naming explicitly. **Authenticity** means we know who (or what) we are talking to, whether that is a user, a partner's webhook, or a signed software package. **Accountability** means that important actions can be traced to whoever performed them, which is why audit logging appears in Chapter 23. **Privacy** overlaps with confidentiality but goes further: it covers collecting only the data you need, using it only for its stated purpose, and deleting it when it is no longer required.

When you evaluate a feature, it helps to ask how each of these properties could fail. The grades bug above is a confidentiality failure. A quiz endpoint that trusts a score sent from the browser would be an integrity failure. An unlimited "import course from URL" feature that fetches gigabytes of data would be an availability failure, and possibly worse, as Chapter 18 explains.

Why the Job Lands on Developers

Many organizations have a security team, and some developers reasonably wonder why they should carry this responsibility themselves. There are several practical reasons, none of which depend on scare stories.

The weaknesses live in the code and the design

Perimeter defenses such as network firewalls and web application firewalls (WAFs) can block some known attack patterns, but they cannot understand your business rules. A WAF has no idea that student 4182 should not see student 4183's grades, that a coupon should be usable only once, or that an instructor in one organization must never edit a course owned by another. Those rules exist only in your requirements and your code. If the code does not enforce them, nothing else will.

Developers make hundreds of security decisions a week

Every time you choose how to build a SQL query, where to store a token, which fields an API accepts, what an error message says, or which package to install, you are making a security decision, whether or not you think of it that way. A security team cannot review each of those decisions in real time. The only people present at the moment of decision are the people writing the code.

Fixing early is cheaper and calmer

A flaw caught while designing a feature costs a conversation. The same flaw caught in code review costs a revision. Caught in production, it can cost an incident response, customer communication, emergency patches, and sometimes legal obligations. We will not attach invented numbers to this; you already know from ordinary bugs that late fixes are more disruptive than early ones, and security bugs follow the same pattern with added pressure. This is the idea behind "shifting left": moving security activities earlier in the development lifecycle, as Chapter 25 describes.

Modern delivery leaves no gap for a separate security phase

When Lattice Learn deploys to production many times a day through an automated pipeline, there is no multi-week "security testing phase" before release. Security has to be built into the same activities that already exist: design discussions, pull requests, automated tests, dependency updates, and deployment configuration. That puts it into developers' hands by default.

> **NOTE**
> Saying that security is a developer's job does not mean developers must become full-time security specialists or carry the responsibility alone. It means security is one of the quality attributes you own, alongside correctness, performance, and maintainability, with specialists supporting you through training, tooling, reviews, and incident response.

Meet Lattice Learn

Security advice is easiest to absorb when it is attached to something concrete. Throughout this book we will work on **Lattice Learn**, a fictional online learning platform that is realistic enough to contain the same kinds of weaknesses real products contain. It is not modeled on any real company, and every vulnerability we discuss in it is intentional and illustrative.

Users and roles

Role	What they do	Typical sensitive actions
Student	Takes courses, quizzes, uploads assignments	View own grades, post in forums
Instructor	Creates courses, grades work	Edit grades, publish lessons
Organization Admin	Manages a B2B tenant	Invite users, configure SSO, billing
Support Agent	Helps users with problems	View accounts, reset access
Platform Admin	Operates the whole platform	Admin console, feature flags

Organizations are **tenants**: a company or university buys seats for its people, and its data must be isolated from every other organization's data. Multi-tenancy is a recurring theme, especially in Chapter 6.

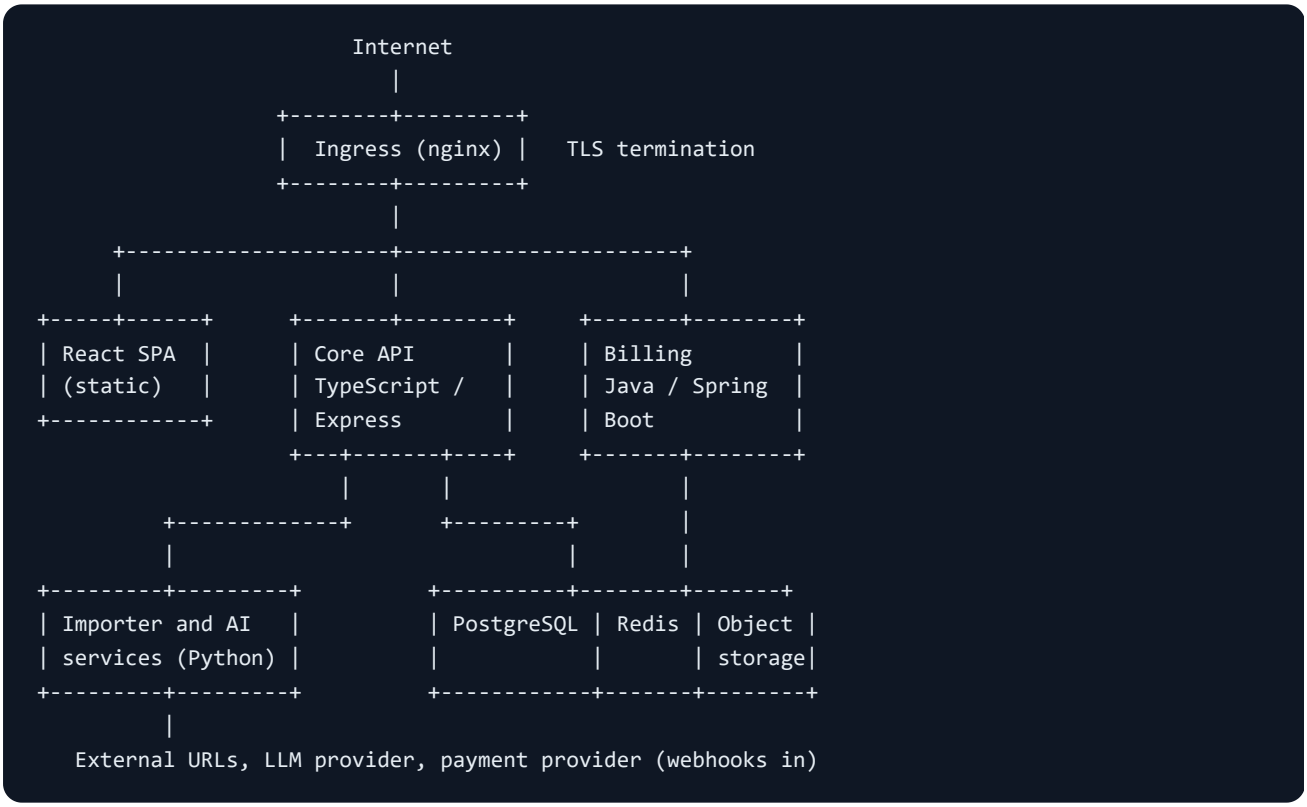
Features

Lattice Learn offers a course catalog with search, video lessons, quizzes and grades, assignment file uploads, and discussion forums that accept rich text. Students who finish a course receive a PDF certificate. Payments and subscriptions go through a fictional payment provider, which notifies Lattice Learn of events through webhooks. Partners integrate through a public REST API authenticated with API keys, and organizations can sign their users in through single sign-on using OAuth 2.0 and OpenID Connect. Instructors can import course material from a URL, and students can ask questions of an AI study assistant. Staff manage everything through an admin console.

Almost every one of those features maps to a chapter of this book, because almost every feature introduces a way things can go wrong. Rich text in forums brings cross-site scripting (Chapter 12). File uploads and PDF generation bring the risks of Chapter 14. Webhooks raise questions of authenticity and integrity (Chapters 19 and 22). Importing from a URL is the textbook setting for server-side request forgery (Chapter 18). The AI assistant introduces prompt injection and data leakage (Chapter 28).

Architecture

The platform uses a common, modern stack. The main API is written in TypeScript with Express and serves a React single-page application. A few services are written in Python with FastAPI, including the course importer and the AI assistant backend. The billing service is written in Java with Spring Boot. Data lives in PostgreSQL, sessions and rate-limit counters live in Redis, and uploaded files and generated certificates live in object storage. Everything runs in containers on Kubernetes.



We will come back to this diagram often. In Chapter 2 we will mark where data crosses from less-trusted to more-trusted parts of the system, and in Chapter 3 we will turn it into a data flow diagram for threat modeling.

The Attacker Is Usually Just a User With Different Goals

Popular culture imagines attackers as hooded figures typing furiously. In practice, many of the people who find and abuse weaknesses in web applications are ordinary users, automated bots, or opportunists who noticed something odd. For Lattice Learn, realistic adversaries include:

- A student who wants better grades or free access to paid courses.
- A user of one organization who is curious about another organization's data.
- Automated bots that try leaked username and password pairs against every login page they can find, the credential stuffing described in Chapter 7.
- Spammers who want to post links in forums or abuse the AI assistant for free computation.
- A former employee of a customer organization whose access was never revoked.
- Someone who has compromised a third-party package that Lattice Learn depends on, as discussed in Chapters 21 and 22.

None of these adversaries needs to be a genius. Most rely on the application trusting something it should not: a value in a URL, a field in a JSON body, a claim in a token, a header added by a client, or a package pulled from a registry. Once you start looking for misplaced trust, you will find it everywhere, and that is a good thing, because each instance you find is one you can fix.

Security as a Quality Attribute

It helps to treat security the same way you already treat other quality attributes. You do not add performance at the end of a project by running a single benchmark; you design data access sensibly, measure regularly, and fix regressions as they appear. Security works the same way:

- **Requirements.** Features come with security acceptance criteria, such as "an instructor can only grade submissions for courses they teach." Chapter 4 shows how OWASP ASVS helps you write these.
- **Design.** Risky features get a lightweight threat model before implementation (Chapter 3).
- **Implementation.** Secure defaults, safe libraries, and patterns that make the right thing easy.
- **Verification.** Unit and integration tests for authorization rules, plus automated scanners and security-focused review (Chapter 24).
- **Operation.** Logging, monitoring, patching, and a plan for when something goes wrong (Chapter 23).

Seen this way, a security bug is simply a bug whose consequences involve someone doing something they should not be able to do. It belongs in the same tracker, gets the same root-cause analysis, and deserves the same regression tests as any other defect.

Fixed — test/grades.authz.test.ts

```
import request from "supertest";
import { app } from "../src/app";
import { loginAs, seed } from "./helpers";

describe("GET /students/:id/grades authorization", () => {
  it("lets a student read their own grades", async () => {
    const { alice } = await seed();
    const agent = await loginAs(alice);
    await agent.get(`/students/${alice.id}/grades`).expect(200);
  });

  it("hides another student's grades", async () => {
    const { alice, bob } = await seed();
    const agent = await loginAs(alice);
    await agent.get(`/students/${bob.id}/grades`).expect(404);
  });

  it("rejects anonymous requests", async () => {
    const { alice } = await seed();
    await request(app).get(`/students/${alice.id}/grades`).expect(401);
  });
});
```

The second test is the important one. It encodes the abuse case — "a student tries to read another student's grades" — as an executable requirement. If someone later refactors the route and drops the policy check, the build fails. Tests like this are among the highest-value security controls you can write, and they need no specialist tools.

Principles That Hold Everything Together

Specific vulnerabilities come and go as frameworks evolve, but a small set of design principles stays useful across languages and decades. Many trace back to classic work on protecting information in computer systems; you will see them applied repeatedly in this book.

Never trust input from outside a trust boundary

Anything that arrives from a client — URL parameters, headers, cookies, JSON bodies, uploaded files — can be set to any value by whoever controls the client. The same is true of data from partner systems, webhooks, and URLs you fetch. Validate it, constrain it, and treat it as data rather than instructions. Chapter 2 defines trust boundaries carefully.

Deny by default

When a decision is uncertain, the safe answer is "no." Routes should require authentication unless they are explicitly public; permissions should be granted, not assumed. The fixed grades route denies unless a policy says otherwise.

Least privilege

Every user, service, token, and database account should have only the access it needs. The course importer does not need write access to the grades table; the AI assistant does not need to see payment data; a CI job that runs tests does not need production deployment credentials.

Defense in depth

Assume any single control can fail and layer several independent ones. Output encoding prevents most XSS, and a Content Security Policy limits the damage if encoding is missed somewhere (Chapter 12). Authorization checks in the API protect data, and row-level filtering by tenant in queries adds a second barrier (Chapter 6).

Fail securely

Errors happen. When they do, the system should end up in a safe state: access denied rather than granted, a generic error message rather than a stack trace, a transaction rolled back rather than half-applied.

Vulnerable — importer/app/permissions.py

```
from fastapi import HTTPException

async def ensure_can_import(user, course_id: str, authz_client) -> None:
    try:
        allowed = await authz_client.check(user.id, "course:import", course_id)
    except Exception:
        # The policy service is slow sometimes; don't block instructors.
        allowed = True
    if not allowed:
        raise HTTPException(status_code=403, detail="forbidden")
```

Fixed — importer/app/permissions.py

```
import logging
from fastapi import HTTPException

log = logging.getLogger("importer.authz")

async def ensure_can_import(user, course_id: str, authz_client) -> None:
    try:
        allowed = await authz_client.check(user.id, "course:import", course_id)
    except Exception:
        # Fail closed: an unavailable policy service must never grant access.
        log.warning("authz check failed", extra={"user_id": user.id,
```

```
                                                "course_id": course_id})
        raise HTTPException(status_code=503, detail="try_again_later")
    if not allowed:
        raise HTTPException(status_code=403, detail="forbidden")
```

The vulnerable version was written with good intentions — nobody wants instructors blocked by a flaky dependency — but it turns any outage of the policy service into a bypass of every authorization rule. The fixed version fails closed, logs the event for operators, and returns a retryable error. If availability of the policy service is a real concern, the answer is to make it more reliable or to cache recent decisions carefully, not to grant access on failure.

Minimize the attack surface

Every endpoint, parameter, open port, debug feature, and installed package is something that can be abused. Remove what you do not use, disable what you do not need in production, and keep administrative functions separate from public ones.

Keep it simple and use proven components

Complex security logic hides mistakes. Prefer well-maintained libraries for cryptography, authentication protocols, and parsing rather than inventing your own, and centralize security decisions — such as authorization policies — so they are written once and reviewed carefully.

> **TIP**
>
> When you review a pull request, pick one principle and apply it deliberately. "What input crosses a trust boundary here?" or "What happens if this call throws?" are quick questions that catch a surprising number of problems without slowing the review down.

Common Myths That Get in the Way

Several comfortable beliefs keep teams from taking ownership of application security. Each contains a grain of truth, which is what makes it persuasive.

"Our framework handles security for us"

Modern frameworks do help a great deal. React escapes text by default, ORMs parameterize queries, and many frameworks ship with CSRF protection. But frameworks only protect you when you use them as intended. React's escaping disappears the moment someone uses `dangerouslySetInnerHTML` on forum content; an ORM's protection vanishes when a developer concatenates strings into a raw query; and no framework knows your authorization rules. Frameworks raise the floor; they do not replace understanding.

"We are behind a firewall, so internal services are safe"

Internal services are reachable by anything that can make requests from inside the network, including your own application when it is tricked into doing so. Server-side request forgery (Chapter 18) exists precisely because "internal" is not the same as "trusted." Modern architectures increasingly assume that network location alone grants no trust, an idea often called zero trust.

"Nobody would bother attacking us"

Much malicious traffic on the internet is automated and indiscriminate. Bots scan for exposed admin panels, default credentials, known-vulnerable software versions, and leaked keys without caring whose system they have found. A small platform is not too uninteresting to be targeted; it is simply targeted by automation rather than by a person.

"Security slows delivery down"

Badly integrated security can slow things down: late-stage reviews that block releases, scanners that produce floods of false positives, and rules nobody understands. Well-integrated security tends to have the opposite effect. Secure defaults, reusable patterns, and automated checks remove whole classes of decisions from each pull request, and fewer incidents means fewer disruptive emergencies. Much of this book is about making the secure path the easy path.

Who Else Is Involved

Developers are central, but they are not alone. On a healthy team, security responsibilities are shared:

Role	Security contribution
Developers	Secure design and code, tests, fixing findings
Tech leads	Architecture decisions, review standards, priorities
Product managers	Security and privacy requirements, abuse cases
DevOps and platform	Pipelines, secrets, cluster and cloud hardening
Security team	Guidance, tooling, reviews, testing, incident response
QA	Negative tests, authorization test matrices

Many organizations also run a **security champions** program, in which a developer on each team takes on extra security training and becomes the first point of contact for security questions. Champions do not replace the security team; they connect it to day-to-day development. If your organization lacks such a program, reading this book and sharing what you learn is a reasonable first step toward becoming one.

Ethics, Authorization, and the Local Lab

To defend a system, you need to understand how it can be attacked. This book therefore shows attack inputs, but only at the level needed to understand and test a defense, and only against Lattice Learn running on your own machine.

> **WARNING**
>
> Only perform security testing on systems you own or are explicitly authorized, in writing, to test. Probing a website, API, or network without permission can break laws and contracts even if your intentions are good and no damage results. Bug bounty and vulnerability disclosure programs define exactly what is in scope; read and follow their rules. Everything in this book is meant for your own code and your own lab.

A local lab lets you experiment freely. The following Docker Compose file starts the pieces of Lattice Learn used in early chapters. It binds every port to the loopback interface only, so the deliberately vulnerable services are not reachable from your network.

Lab — lab/docker-compose.yml

```
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_DB: lattice
      POSTGRES_USER: lattice
      POSTGRES_PASSWORD: lab-only-password   # lab only; never reuse
    ports:
      - "127.0.0.1:5432:5432"
    volumes:
      - ./seed:/docker-entrypoint-initdb.d:ro

  redis:
    image: redis:7
    ports:
      - "127.0.0.1:6379:6379"

  api:
    build: ../api
    environment:
      NODE_ENV: development
      DATABASE_URL: postgres://lattice:lab-only-password@db:5432/lattice
      REDIS_URL: redis://redis:6379
      LAB_MODE: "vulnerable"     # switch to "fixed" to compare behavior
    ports:
      - "127.0.0.1:3000:3000"
    depends_on: [db, redis]

  web:
    build: ../web
```

```
    ports:
      - "127.0.0.1:5173:5173"
    depends_on: [api]
```

The `LAB_MODE` variable lets the lab switch between vulnerable and fixed versions of the routes discussed in each chapter, so you can see a weakness, apply the defense, and confirm the behavior changes. The password is deliberately labeled as lab-only; real credentials never belong in a Compose file or a repository, as Chapter 16 explains.

Lab — lab/check-grades.sh

```
#!/usr/bin/env sh
# Verifies the grades authorization fix in the local lab only.
set -eu
BASE="http://127.0.0.1:3000"

# Log in as the seeded student "alice" and keep the session cookie.
curl -s -c /tmp/alice.txt -H "Content-Type: application/json" \
  -d '{"email":"alice@lab.test","password":"lab-alice"}' \
  "$BASE/auth/login" > /dev/null

# Alice requests Bob's grades (seeded student id 1002). Expect 404 when fixed.
code=$(curl -s -o /dev/null -w "%{http_code}" -b /tmp/alice.txt \
  "$BASE/students/1002/grades")
echo "Cross-student grades request returned HTTP $code"
```

This script is not an attack tool; it is a regression check. It uses seeded lab accounts on a `.test` domain that cannot exist on the public internet, and it targets only the loopback address. Running it in vulnerable mode returns `200`; in fixed mode it returns `404`. That before-and-after pattern repeats throughout the book.

How This Book Is Organized

The book follows the **OWASP Top 10 2021** as its primary framework. The Top 10 is an awareness document produced by the Open Worldwide Application Security Project (OWASP), a nonprofit community that publishes free, openly licensed guidance. The 2021 edition's categories are A01 Broken Access Control, A02 Cryptographic Failures, A03 Injection (which includes cross-site scripting), A04 Insecure Design, A05 Security Misconfiguration, A06 Vulnerable and Outdated Components, A07 Identification and Authentication Failures, A08 Software and Data Integrity Failures, A09 Security Logging and Monitoring Failures, and A10 Server-Side Request Forgery. OWASP periodically updates the list; check owasp.org for the latest edition and how its categories map to the 2021 list. The fundamentals you learn here carry over regardless of how categories are renamed or regrouped.

The book is divided into six parts: