

SECURITY COURSE

OWASP Top 10

10 Chapters

Philip Osztromok

Generated with Claude

OWASP TOP 10

Contents

Chapter 1	Access Control	2
Chapter 2	Crypto Failures	7
Chapter 3	Injection	11
Chapter 4	Insecure Design	15
Chapter 5	Misconfiguration	19
Chapter 6	Outdated Components	24
Chapter 7	Auth Failures	28
Chapter 8	Integrity Failures	32
Chapter 9	Logging Failures	37
Chapter 10	Server-Side Forgery	42

OWASP Top 10

Course 1 · Chapter 1 · A01: Broken Access Control

Welcome to a survey of the **OWASP Top 10** — the web-security field's most widely-used map of the risks that matter most. Where the HTTPS, Authentication, CSRF, XSS, and SQL-injection courses each went *deep* on one problem, this course goes *wide*: ten categories that, together, cover most of what goes wrong in real applications — and where each one connects to the deep-dive you may already have done. We start at number one.

First: What the OWASP Top 10 Actually Is

The **OWASP Top 10** is a periodically-updated *awareness document* published by the Open Worldwide Application Security Project, ranking the most critical web-application security risks. A few things to understand up front so you read it correctly:

- **It's a risk ranking, not a checklist or standard.** It tells you what to *prioritize*, not a complete list of everything to test. (OWASP's *ASVS* is the verifiable standard; the Top 10 is the awareness layer.)
- **The categories are broad themes**, each grouping many specific weaknesses (CWEs), not single bugs.
- **It's data-driven plus a community survey** — built from real-world vulnerability data across many organizations, with the current edition being **2021**.
- **The ranking shifts over time** as the field changes — which is itself informative (Injection, long at #1, dropped to #3 in 2021 as frameworks improved; Broken Access Control rose to #1).

> **THIS COURSE IS THE MAP; YOUR OTHER COURSES ARE THE TERRITORY**
>
> Several Top 10 categories *are* the subjects of your deep-dive courses: **A02 Cryptographic Failures** → HTTPS & password storage, **A03 Injection** → SQLi & XSS, **A07 Authentication Failures** → the Auth course, **A10 SSRF** → the CSRF course's SSRF section. For those, this course gives the *overview and where-it-fits* and points you back to the depth. The real new ground is the categories you *haven't* covered — starting right here with the one that tops the list.

A01: Broken Access Control — The #1 Risk

Access control enforces *what an authenticated user is allowed to do* — it's **authorization** (the "what may you do?" from the Auth course, distinct from authentication's "who are you?"). **Broken access control** means those restrictions aren't properly enforced, so users can act *outside their intended permissions*: read others' data, modify records they shouldn't, or reach admin functionality. In 2021 it became the **#1 category** — found in a striking share of tested applications.

The Common Forms

FORM	WHAT HAPPENS
IDOR (Insecure Direct Object Reference)	change an ID in the request (`/account?id=124` → `123`) and access another user's object
Vertical privilege escalation	a normal user reaches admin-only functionality (`/admin` , an admin API)
Horizontal privilege escalation	a user accesses *another user's* data at the same privilege level
Missing function-level access control	a sensitive endpoint isn't protected server-side — just hidden in the UI
Forced browsing	guessing/visiting unlinked URLs (`/admin/deleteUser`) that lack checks
Metadata/token tampering	editing a cookie, JWT claim, or hidden field (`role=user` → `admin`)

```
// IDOR — the endpoint trusts the id without checking ownership
// GET /api/invoices/1043   (you're allowed to see YOUR invoice 1043)
// GET /api/invoices/1044   → returns someone ELSE'S invoice. Broken access control.
```

Why It Tops the List

Two reasons. First, it's **extremely common**: access checks must be applied to *every* protected object and action, and missing just one is a hole — a large surface to get perfectly right. Second, the **impact is severe**: it leads directly to data breaches (reading every user's records by walking

IDs), account takeover, and unauthorized administrative actions. It's also often *trivial to exploit* — no special tools, just changing a number or a role value in a request.

> **THE ROOT MISTAKE: TRUSTING THE CLIENT, OR RELYING ON "THEY CAN'T SEE THE LINK"**
>
> Broken access control almost always comes from enforcing access in the wrong place — or not at all. **Hiding a button in the UI is not access control** (the attacker calls the API directly). **Trusting a client-supplied value** — a `role` field in the request, an `isAdmin` cookie, an ID the user can change — is not access control (the client is attacker-controlled). And **"security through obscurity"** — assuming an unlinked admin URL is safe because it's not in the menu — fails to forced browsing. Every access decision must be made *server-side*, based on the *authenticated session's* identity and permissions, on *every* request — never on anything the client sends or sees.

The Defences

- **Deny by default** — everything is forbidden unless explicitly granted; new endpoints are protected automatically, not accidentally public.
- **Enforce server-side, every request** — never rely on hidden UI, disabled buttons, or client-sent role/ID values.
- **Check ownership on every object access** — "does *this* user own/has rights to *this* record?", not just "is the user logged in?" (the IDOR fix).
- **Centralize the access-control logic** — one well-tested mechanism (middleware, a policy layer) rather than ad-hoc checks scattered and forgotten.
- **Use a model** — RBAC (roles) or ABAC (attributes) applied consistently; least privilege throughout.
- **Make IDs hard to enumerate** (defence in depth) — random UUIDs over sequential integers — but this *supplements* ownership checks, never replaces them.
- **Log access-control failures and alert** — repeated denials are a strong attack signal (ties to A09).

Hands-On Exercises

EXERCISE 1

Classify each as the specific broken-access-control form (IDOR, vertical escalation, horizontal escalation, missing function-level control, forced browsing, token tampering): (a) changing `?userId=500` to read another profile; (b) a standard user POSTing to `/admin/api/deleteUser`; (c) editing a JWT's `role` claim to `admin`; (d) visiting an unlinked `/reports/all` that has no check.

View solution

EXERCISE 2

Given an IDOR-vulnerable endpoint `GET /api/invoices/:id` that returns any invoice by ID, explain the attack and write the server-side ownership-check fix. Explain why using random UUIDs for IDs is helpful but not a substitute for the check.

View solution

EXERCISE 3

Explain "deny by default" and why access control must be enforced server-side on every request. Then rebut three bad arguments: "the button is hidden for non-admins," "we check `isAdmin` from the cookie," and "that URL isn't linked anywhere." State the correct principle for each.

View solution

CHAPTER 1 QUICK REFERENCE

- **OWASP Top 10** = a periodically-updated *risk-awareness ranking* (not a checklist/standard); current edition **2021**; data-driven + community survey
- This course is the **map**; A02/A03/A07/A10 point back to your HTTPS/SQLi/XSS/Auth/CSRF deep-dives
- **A01 Broken Access Control** = the #1 risk (2021): authorization isn't enforced, so users act outside their permissions
- Forms: **IDOR** · vertical escalation · horizontal escalation · missing function-level control · forced browsing · token/metadata tampering
- Root mistake: **trusting the client** (role/ID values, hidden UI) or **obscurity** (unlinked URLs) instead of server-side checks
- Defences: **deny by default** · enforce **server-side every request** · **check ownership** per object (IDOR fix) · centralize · RBAC/least privilege · log failures

- Random UUIDs help (harder to enumerate) but **never replace** ownership checks
- **Next chapter:** A02 Cryptographic Failures — weak/missing encryption, plaintext, bad TLS & hashing (→ HTTPS & Auth courses)

OWASP Top 10

Course 1 · Chapter 2 · A02: Cryptographic Failures

Number two on the list is about protecting **sensitive data** — and what goes wrong when the cryptography that's supposed to protect it is weak, misused, or simply absent. This is a category where your existing courses do much of the heavy lifting: it's largely "the **HTTPS** course (data in transit) plus the password-storage chapter of the **Auth** course (data at rest & hashing), generalized." This chapter maps the whole category and points you to that depth.

A02: Cryptographic Failures — and a Telling Rename

In 2021 this category was **renamed from "Sensitive Data Exposure" to "Cryptographic Failures."** That rename matters: the old name described the *symptom* (data got exposed), the new one names the *root cause* (the cryptography failed or was missing). It's a reminder that exposed data is almost always a downstream result of a crypto mistake — so you fix it by getting the cryptography right, not by chasing leaks. It ranks **#2**.

Step One: Know What's Sensitive

You can't protect data correctly until you know *which* data needs protecting and *what kind*. Before any crypto, classify your data and its requirements:

- **What is sensitive?** — passwords, payment/card data, health records, personal data (PII), session tokens, API keys, anything with legal/regulatory weight (GDPR, PCI-DSS, HIPAA).
- **In transit, at rest, or both?** — data moving over the network needs transport protection; data stored needs at-rest protection; most sensitive data needs both.
- **Does it need to be recoverable?** — this decides *hashing vs encryption* (below) — the single most common crypto-choice mistake.

The Common Cryptographic Failures

FAILURE	EXAMPLE	DEEP-DIVE
`Cleartext transmission`	plain HTTP, unencrypted email/DB connections	HTTPS course
`Weak / outdated algorithms`	MD5, SHA-1, DES, RC4, AES-ECB mode	HTTPS Ch. 8
`Weak password hashing`	fast hashes (MD5/SHA-256) instead of Argon2/bcrypt	Auth Ch. 2
`Encrypting what should be hashed`	storing reversible passwords "encrypted"	Auth Ch. 2
`Poor key management`	hardcoded/default keys, keys in source control	this chapter
`No encryption at rest`	plaintext PII / card data in the database	this chapter
`Weak randomness`	`Math.random()` for tokens/keys (not a CSPRNG)	Auth Ch. 4
`Improper cert validation`	disabling TLS verification "to make it work"	HTTPS Ch. 5/11

The Most Common Mistake: Hashing vs Encryption

The crypto-choice error that appears again and again: using the *wrong primitive* for the data. The rule from the Auth course generalizes here:

> **HASH WHAT YOU ONLY NEED TO VERIFY; ENCRYPT WHAT YOU NEED TO READ BACK**
>
> **Encryption is reversible** (with a key) — use it for data you must *recover* later (a stored credit-card number you'll charge again, a document you'll display). **Hashing is one-way** — use it for data you only need to *verify*, never recover: **passwords** are the canonical case (you check a login by hashing the attempt and comparing — you never need the original back). Storing passwords *encrypted* is a classic A02 failure, because the key becomes a single point of total compromise: steal the key, decrypt every password. And the hash must be a *slow, salted* password hash (Argon2id/bcrypt), not a fast general-purpose one. "We encrypt passwords" in a breach statement is a red flag, not reassurance.

The Defences

- **Classify data first** — know what's sensitive and its in-transit/at-rest/regulatory needs; don't store sensitive data you don't need.

- **Encrypt in transit** — TLS everywhere (HTTPS + HSTS), including internal and DB connections — the entire HTTPS course.
- **Encrypt at rest** — sensitive fields/databases encrypted with strong, modern algorithms (**AES-GCM** and similar AEAD modes).
- **Hash passwords correctly** — slow, salted **Argon2id / bcrypt**; never fast hashes, never "encryption" (Auth Ch. 2).
- **Manage keys properly** — never hardcode keys or commit them to source control; use a secrets manager / KMS; rotate keys; separate keys from the data they protect.
- **Use a CSPRNG** for tokens/keys/IVs — `crypto.randomBytes`, never `Math.random()`.
- **Validate TLS properly** — never disable certificate verification; pin algorithm versions (no SSL/TLS 1.0/1.1, no weak ciphers).
- **Don't cache sensitive data** — appropriate cache-control headers so secrets aren't stored by browsers/proxies.

THE GOLDEN RULE: DON'T ROLL YOUR OWN CRYPTO

The most reliable cryptographic-failure prevention is simply **not implementing cryptography yourself.** Cryptographic primitives are devastatingly easy to get subtly wrong (a reused IV, an ECB mode, a non-constant-time comparison, a homemade "encryption" that's really encoding) — and the failure is usually invisible until exploited. Use *vetted, maintained libraries* and high-level constructs (your platform's TLS stack, `libsodium`/the language's standard crypto, a maintained password-hashing library, a KMS), configured per current guidance. Reserve "implementing crypto" for specialists; for everyone else, the win is choosing the right well-built tool and using it correctly — exactly the lesson from HTTPS and Auth.

Hands-On Exercises

EXERCISE 1

For each, state the protection needed (encrypt in transit / encrypt at rest / hash) and why: (a) a user's login password; (b) a credit-card number you must charge monthly; (c) data sent from the browser to your API; (d) session tokens stored in the database. Note any that need more than one.

View solution

EXERCISE 2

Spot every cryptographic failure in this design and fix each: passwords stored as `MD5`; the site served over `HTTP`; the encryption key hardcoded in the source and committed to git; session IDs generated with `Math.random()`. Map each to its A02 sub-category.

View solution

EXERCISE 3

Explain the hashing-vs-encryption decision: which to use for passwords vs for a recoverable card number, and why storing passwords "encrypted" is a failure. Then explain why "don't roll your own crypto" is the strongest single piece of A02 advice, with two ways homemade crypto goes wrong.

View solution

CHAPTER 2 QUICK REFERENCE

- **A02 Cryptographic Failures** (#2) — sensitive data unprotected because crypto is weak, misused, or absent; **renamed** from "Sensitive Data Exposure" to name the *cause*, not the symptom
- **Classify first** — know what's sensitive (passwords, PII, cards, tokens) and whether it needs protection in transit, at rest, or both
- Common failures: **cleartext (HTTP)** · weak algorithms (MD5/SHA-1/DES/ECB) · weak password hashing · encrypting-what-should-be-hashed · poor key management · no at-rest encryption · weak randomness · bad cert validation
- **Hash vs encrypt**: hash what you only verify (passwords → Argon2id/bcrypt, salted/slow); encrypt what you must read back (cards) — storing passwords "encrypted" is a failure
- Defences: classify · TLS in transit (HTTPS+HSTS) · AES-GCM at rest · proper key management (no hardcoded keys/secrets in git, use a KMS) · CSPRNG · valid TLS · no caching secrets
- **Don't roll your own crypto** — use vetted libraries/high-level constructs; homemade crypto fails subtly and invisibly
- Deep-dives: **HTTPS** (in transit), **Auth Ch. 2** (password hashing / at rest)
- **Next chapter:** A03 Injection — SQLi, XSS & the data-vs-code family (→ SQLi & XSS courses)

OWASP Top 10

Course 1 · Chapter 3 · A03: Injection

If you've done the SQL-injection and XSS courses, this chapter is a homecoming: **A03 Injection** is the family those two belong to. It's the one category where you've already gone deep *twice* — so here we zoom out to see the whole family, the single root cause they share, and the unified defence, then point you back to the depth.

A03: Injection — Once #1, Now #3

Injection sat at **#1 for years** and dropped to **#3 in 2021** — and that movement tells a story. It didn't get less dangerous; rather, frameworks made the safe path the default (ORMs parameterize, template engines auto-escape), so the *rate* of injection fell. The 2021 category also **absorbed Cross-Site Scripting (XSS)**, which used to be its own entry — formally recognizing that XSS is just injection aimed at the browser. So A03 now spans the whole family.

The One Root Cause

EVERY INJECTION IS THE SAME BUG: UNTRUSTED INPUT PARSED AS CODE

SQLi, XSS, command injection, LDAP injection, template injection, NoSQL injection — all are **one mistake**: untrusted data crosses a boundary where an interpreter parses it as *code* instead of keeping it as *data*. The interpreter differs (a SQL database, the browser, the OS shell, an LDAP server, a template engine), and so does the breakout syntax, but the flaw is identical. This is the unifying idea of the whole SQLi and XSS courses — "data vs code" — and A03 is simply that idea given a Top 10 slot. Learn it once and every member of the family makes sense.

The Members of the Family

INJECTION TYPE	INTERPRETER	BREAKOUT / EXAMPLE	DEEP-DIVE
SQL injection	SQL database	`' OR '1'='1`	SQLi course
Cross-Site Scripting (XSS)	the browser	`<script>…</script>`	XSS course

INJECTION TYPE	INTERPRETER	BREAKOUT / EXAMPLE	DEEP-DIVE
OS command injection	the OS shell	`; rm -rf /` / `&& curl evil`	this chapter
LDAP injection	an LDAP directory	`*)(uid=*` filter manipulation	this chapter
NoSQL injection	a document DB	`{"$ne": null}` operator injection	SQLi Ch. 9
Template injection (SSTI)	a template engine	`{{7*7}}` → `49`	this chapter

The One You Haven't Covered: OS Command Injection

Since SQLi and XSS have their own courses, the family member worth a closer look here is **command injection** — when untrusted input reaches an OS shell command. It's especially dangerous because it can mean full *server* compromise (remote code execution), not just data access:

```
// VULNERABLE — user input concatenated into a shell command
const cmd = "ping -c 1 " + req.query.host;   // host = "8.8.8.8; rm -rf /"
exec(cmd);   // runs: ping -c 1 8.8.8.8; rm -rf /   ← the ; chains a second command

// SAFE — pass arguments as a separate array; no shell string to inject into
execFile("ping", ["-c", "1", host]);   // host is an argument, never parsed as shell
```

Notice the fix is structurally identical to parameterized SQL queries: **separate the command from the data**. Passing arguments as an array (rather than building one shell string) means the input can never become shell syntax — the same "keep data off the code channel" principle, in a different interpreter.

Spot-the-Pattern: SSTI

Server-Side Template Injection is a modern family member worth recognizing: if untrusted input is rendered *as a template* (not just *into* one), the template engine evaluates it. The classic probe is `{{7*7}}` — if the page returns `49`, the input is being evaluated as template code, which often escalates to remote code execution. The cause is the same (input treated as code); the fix is to never render user input as a template — pass it as *data* to a fixed template instead.

The Unified Defence

Because all injection shares one root cause, the defences rhyme across the family — and they're exactly what the SQLi and XSS courses taught:

INTERPRETER	KEEP-DATA-AS-DATA DEFENCE
SQL	parameterized queries / prepared statements
Browser (XSS)	context-aware output encoding (+ CSP)
OS shell	avoid the shell; pass args as an array (`execFile`)
LDAP / NoSQL	parameterized/safe APIs; validate types & escape filter metachars
Templates	never render user input as a template; pass it as data

THE SINGLE PRINCIPLE ACROSS THE WHOLE CATEGORY

Keep untrusted input as data; never let it become code — by separating the command/query/markup from the data on different channels. Parameterized queries (SQL), argument arrays (shell), output encoding (HTML), safe APIs (LDAP/NoSQL), data-not-template rendering (SSTI) are all one idea in different costumes. Supporting layers help too — allowlist input validation, least privilege (a command-injection on a low-privileged account does less), and avoiding dangerous sinks (don't call the shell at all if you can use a library) — but as in SQLi/XSS, those are defence in depth; the load-bearing fix is keeping data off the code channel.

Hands-On Exercises

EXERCISE 1

For each, name the injection type, the interpreter, and the unified defence: (a) `' UNION SELECT …` in a search box; (b) `<img src=x onerror=…>` in a comment; (c) `; cat /etc/passwd` in a hostname field; (d) `{{7*7}}` returning `49`. State the one root cause they all share.

View solution

EXERCISE 2

Given `exec("ping -c 1 " + req.query.host)`, show how an attacker achieves command execution and why it can mean full server compromise. Write the safe version and explain why it's the same principle as parameterized SQL queries.

View solution

EXERCISE 3

Explain why A03 moved from #1 to #3 and absorbed XSS, and why that doesn't mean injection is "solved." Then state the one principle unifying SQL, XSS, command, LDAP, NoSQL, and template injection, and give the keep-data-as-data defence for three of them.

View solution

CHAPTER 3 QUICK REFERENCE

- **A03 Injection** — once #1, now #3 (frameworks made safe defaults common); 2021 **absorbed XSS** as injection-into-the-browser
- **One root cause**: untrusted input parsed as *code* instead of staying *data* — same bug across the whole family
- Family: **SQLi** (DB), **XSS** (browser), **OS command** (shell), **LDAP**, **NoSQL** (operators), **template/SSTI**
- **Command injection** — input into a shell (`; rm -rf /`) → can be full RCE; fix: avoid the shell, pass args as an array (`execFile`)
- **SSTI** — rendering input *as* a template (`{{7*7}}`→49) → often RCE; fix: pass input as data to a fixed template
- **Unified defence**: keep data off the code channel — parameterized queries (SQL) · output encoding+CSP (XSS) · arg arrays (shell) · safe APIs/type-validation (LDAP/NoSQL) · data-not-template (SSTI)
- Plus defence in depth: allowlist validation, least privilege, avoid dangerous sinks
- Deep-dives: **SQLi course**, **XSS course** (+ SQLi Ch. 9 for NoSQL)
- **Next chapter:** A04 Insecure Design — flaws no code fix solves; threat modeling & secure-by-design

OWASP Top 10

Course 1 · Chapter 4 · A04: Insecure Design

Every category so far has had a concrete fix you can write in code: parameterize the query, hash the password, check ownership. **A04 Insecure Design** is different in kind — it's about flaws baked into the *design itself*, where **no amount of perfect code can save you** because the very thing being built is unsafe. It's the newest category (added in 2021) and arguably the most important shift in how the Top 10 thinks.

A04: Insecure Design — The Category About *What* You Build

The crucial distinction this category introduces: **insecure design vs insecure implementation.**

	IMPLEMENTATION FLAW	DESIGN FLAW (A04)
`What's wrong`	the code does the right thing *wrong*	the *right thing to do* was never specified
`Example`	a query that should be parameterized isn't	no rate limit was ever designed for password reset
`Fixable in code?`	yes — fix the line	no — the missing control must be *designed in*

> **"YOU CANNOT FIX AN INSECURE DESIGN WITH PERFECT IMPLEMENTATION"**
>
> This is the line to remember. If a feature is *designed* without a needed security control, then flawlessly implementing that design just gives you a flawlessly-built insecure feature. Example: a "recover account via security questions" flow. You can implement it with perfect code — parameterized queries, no XSS, great error handling — and it's *still* insecure, because security questions are a weak design (answers are public/guessable, Auth course). The vulnerability isn't a bug; it's the *choice of mechanism*. A04 says: security has to start at **design time**, not be bolted on after the code is written.

What Insecure Design Looks Like

- **Missing controls by design** — no rate limiting on login/reset (enabling brute force/stuffing), no limits on a costly operation, no anti-automation on a high-value action.

- **Trusting the client for business logic** — a checkout that takes the *price* from the request, a "is premium user" flag sent by the browser; the design assumed the client wouldn't lie.
- **Business-logic flaws** — workflows that can be done out of order or abused: applying a discount code unlimited times, refunding more than was paid, skipping a payment step, negative quantities.
- **No defence against abuse of legitimate features** — a coupon system with no per-user cap, a referral program that can be farmed, an API with no quotas.
- **Over-collection / over-exposure of data** — designing to store or return more sensitive data than the feature needs.

```
// A design flaw, not a code bug: trusting the client for the price
// POST /checkout  { "item": "laptop", "price": 1299 }   ← price from the browser
total = req.body.price * qty;   // attacker sends price: 1  → buys a laptop for $1
// No line of code is "wrong" — the DESIGN trusts client-supplied business data.
// Correct design: the SERVER looks up the price from the catalogue by item id.
```

The Core Practice: Threat Modeling

The primary defence against insecure design is **threat modeling** — thinking, *during design*, about what could go wrong and who might attack the feature, before a line is written. A lightweight version asks four questions (the Shostack framework):

1. **What are we building?** — a clear picture of the feature, its data, and its trust boundaries (where data crosses from untrusted to trusted).
2. **What can go wrong?** — brainstorm threats (a useful mnemonic is **STRIDE**: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege).
3. **What are we going to do about it?** — decide which controls to design in (rate limits, server-side validation, authorization checks, abuse caps).
4. **Did we do a good job?** — review and validate the design against the threats.

The payoff is leverage: a control identified at design time is cheap; the same control retrofitted after a breach is expensive and often patchy. A04 essentially says "do this."

Secure-by-Design Principles

A04 promotes a set of design-time principles — most of which you've already met as the *reasons behind* earlier defences:

- **Least privilege** — components and users get the minimum access needed (A01, A02 key management).
- **Defence in depth** — layered controls so one failure isn't catastrophic (the theme of every security course).
- **Fail securely** — on error, default to denying access, not allowing it (deny-by-default, A01).
- **Secure defaults** — the out-of-the-box configuration is the safe one (links to A05).
- **Minimize attack surface** — don't build/expose features and data you don't need.
- **Separation of duties & trust boundaries** — be explicit about what is trusted and validate everything crossing a boundary.

SECURITY AS A PHASE VS SECURITY AS A PROPERTY

The deeper message of A04 is a mindset shift: security is not a *testing phase* bolted on at the end ("we'll pen-test before launch"), it's a *property designed in from the start*. Pen-testing and code review find *implementation* bugs, but they rarely catch a fundamentally insecure *design* — a reviewer sees clean code implementing a flawed idea. The expensive lesson many teams learn is that the cheapest security work happens at the whiteboard: think about abuse cases and threats *before* building, write security requirements alongside functional ones, and use reference architectures/secure patterns. "Shift left" is the industry phrase; A04 is its Top 10 home.

Hands-On Exercises

EXERCISE 1

For each, decide whether it's an insecure *implementation* or an insecure *design*, and justify: (a) a SQL query that forgot to parameterize; (b) a password reset that uses "mother's maiden name"; (c) a checkout that trusts the client-sent price; (d) a session cookie missing the `HttpOnly` flag. Explain why the design ones can't be fixed by better code alone.

View solution

EXERCISE 2

Threat-model a "redeem discount code" feature using the four questions (what are we building / what can go wrong / what do we do / did we do well). Identify at least three design-level abuse cases and the control each needs designed in.

View solution

EXERCISE 3

Explain the principle "you can't fix an insecure design with perfect implementation," with a worked example. Then explain why pen-testing late doesn't catch design flaws, and list four secure-by-design principles with how each would have prevented an earlier-chapter vulnerability.

View solution

CHAPTER 4 QUICK REFERENCE

- **A04 Insecure Design** (new in 2021) — flaws in *what* you build, not how it's coded; **no perfect implementation can fix an insecure design**
- **Design vs implementation**: implementation = doing the right thing wrong (fix the code); design = the right control was never specified (must be designed in)
- Looks like: missing controls (no rate limiting) · trusting client for business logic (client-sent price) · business-logic flaws (unlimited coupon, over-refund) · feature abuse · over-collection
- **Threat modeling** — the core practice: *what are we building? · what can go wrong? (STRIDE) · what do we do? · did we do well?* — done at design time
- **Secure-by-design principles**: least privilege · defence in depth · fail securely · secure defaults · minimize attack surface · explicit trust boundaries
- Mindset shift: security is a **designed-in property**, not a late testing phase — "shift left"; the cheapest security work is at the whiteboard
- Pen-testing finds *implementation* bugs; it rarely catches a flawed *design* (clean code, bad idea)
- **Next chapter:** A05 Security Misconfiguration — default creds, verbose errors, missing hardening

OWASP Top 10

Course 1 · Chapter 5 · A05: Security Misconfiguration

A04 was about flawed designs. **A05 Security Misconfiguration** is about the opposite failure: the design and code can be perfectly secure, but the system is left in an *insecure state* through how it's set up, deployed, or maintained. It's one of the most common categories precisely because modern apps have enormous configuration surfaces — frameworks, servers, cloud services, containers, databases — and any one of them left in an unsafe state is a hole.

A05: Security Misconfiguration — Secure App, Unsafe Setup

Misconfiguration means the application stack is **not hardened**, or is configured insecurely, anywhere across its layers. The code may be flawless; the *configuration* is the vulnerability. This category rose in 2021 as systems grew more complex and more highly configurable (cloud, IaC, microservices) — more knobs means more knobs to get wrong.

The Common Misconfigurations

MISCONFIGURATION	WHY IT'S DANGEROUS
Default credentials	`admin/admin`, default DB/router/CMS passwords left unchanged — instant access
Verbose errors / debug mode	stack traces leak versions, paths, queries (accelerates other attacks)
Unnecessary features enabled	unused ports, services, sample apps, admin consoles — extra attack surface
Open cloud storage	public S3 buckets / blobs exposing data to the whole internet
Missing security headers	no HSTS, CSP, X-Content-Type-Options, etc. — weakens browser protections
Directory listing / exposed files	`.git`, `.env`, backups, config files reachable over HTTP
Overly permissive settings	wildcard CORS, permissive file perms, debug endpoints in prod

MISCONFIGURATION	WHY IT'S DANGEROUS
`Outdated/insecure defaults`	shipped configuration that was never hardened for production

Two Classics Worth Dwelling On

Default Credentials

Shipping or deploying with default/unchanged credentials (`admin/admin`, a database's default root password, a router's factory login) is one of the oldest and still-common breaches. Attackers scan the internet for known default logins; an exposed admin panel with factory credentials is found in minutes, automatically. **Every default credential must be changed before exposure**, and ideally accounts ship disabled or force a password set on first use.

Verbose Errors & Debug Mode in Production

> **A DEBUG STACK TRACE IS A FREE MAP FOR THE ATTACKER**
>
> Leaving detailed error pages or framework debug mode on in production leaks a wealth of intelligence: the exact framework and version (→ A06 known-vuln lookup), file paths and directory structure, database type, and sometimes the failing SQL query or environment variables. It accelerates nearly every other attack (you saw this in the SQLi course — verbose DB errors map the schema). The fix is the recurring pattern: **show users a generic error, log the details server-side.** Debug mode is for development only; shipping it to production is a textbook misconfiguration (real frameworks have leaked secrets this way via their debug consoles).

Security Headers — Cheap, High-Value Hardening

A specific, easy-to-fix slice of A05: HTTP response headers that switch on browser-side protections. Missing them is a misconfiguration; adding them is usually a few lines:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains  # HSTS — force HTTPS
Content-Security-Policy: default-src 'self'; ...                # CSP — limit script sources
X-Content-Type-Options: nosniff                                 # stop MIME-sniffing
X-Frame-Options: DENY  (or CSP frame-ancestors)                 # clickjacking defence
Referrer-Policy: strict-origin-when-cross-origin                # limit referrer leakage
```

These tie back to earlier courses — HSTS and CSP appeared in the HTTPS and XSS courses respectively. From A05's view they're *configuration*: present-and-correct, or a missing hardening

step. Tools like **securityheaders.com** and Mozilla Observatory grade them.

The Defences: Hardening & Repeatability

- **Harden every layer** — OS, web server, app framework, database, cloud services, containers — following each vendor's hardening guidance / a benchmark (CIS Benchmarks).
- **Change all defaults** — credentials, keys, sample content; remove default/sample accounts and apps.
- **Minimize surface** — disable/uninstall unused features, ports, services, and admin endpoints; least functionality.
- **Generic errors in prod, debug off** — detailed errors logged server-side only.
- **Set security headers** — HSTS, CSP, nosniff, frame-ancestors, etc.
- **Lock down storage** — no public buckets unless intended; correct file/object permissions; block access to `.git` / `.env` /backups.

> **THE REAL FIX IS AUTOMATION: CONFIGURATION AS CODE, NOT BY HAND**
>
> Manual configuration drifts — one server gets hardened, the next doesn't; a setting changed for debugging never gets reverted; "secure" environments diverge over time. The durable answer to A05 is **repeatable, automated configuration**: define the hardened state as code (Infrastructure as Code — Terraform/Ansible, hardened container base images, locked-down framework config in version control) so every environment is provisioned identically and securely, and drift is detectable. Add automated **configuration scanning** (cloud posture tools, header checkers, container/image scanners) in CI to catch regressions. A misconfiguration you can re-introduce by hand is one you'll re-introduce; make the secure state the *default, reproducible* state. (This also connects to A08 — pipeline/build integrity.)

Hands-On Exercises

EXERCISE 1

Identify the misconfiguration and its risk in each: (a) a new admin panel still using `admin/admin`; (b) production returning full stack traces; (c) a publicly-readable S3 bucket of user uploads; (d) `.env` reachable at `/.env`. State the fix for each and which other Top 10 category it can amplify.

View solution

EXERCISE 2

List the key security headers (HSTS, CSP, X-Content-Type-Options, frame-ancestors/X-Frame-Options, Referrer-Policy), what each does, and which earlier course/attack it relates to. Explain why "missing headers" counts as a misconfiguration rather than a code bug.

View solution

EXERCISE 3

Explain why manual, per-server configuration is the root of recurring misconfiguration, and how "configuration as code" + automated scanning fixes it. Give a concrete example of configuration drift and how IaC prevents it. Connect this to secure defaults (A04) and pipeline integrity (A08).

View solution

CHAPTER 5 QUICK REFERENCE

- **A05 Security Misconfiguration** — secure code, but the system is left in an insecure *state*; common because the config surface is huge (frameworks, servers, cloud, containers)
- Common: **default credentials** · **verbose errors/debug in prod** · unused features enabled · open cloud storage · missing security headers · exposed `.git` / `.env` /backups · wildcard CORS/permissive settings
- **Default creds** — scanned-for and breached in minutes; change every default before exposure
- **Debug/verbose errors** leak framework version, paths, queries → fuels other attacks; show generic errors, log details server-side
- **Security headers** (HSTS, CSP, nosniff, frame-ancestors, Referrer-Policy) — cheap high-value hardening; missing = misconfiguration
- Defences: harden every layer (CIS Benchmarks) · change defaults · minimize surface · lock down storage · set headers
- **The durable fix is automation** — configuration as code (IaC, hardened images, versioned config) + config scanning in CI, so every environment is identical, hardened, and drift is caught

- **Next chapter:** A06 Vulnerable & Outdated Components — dependency & supply-chain risk

OWASP Top 10

Course 1 · Chapter 6 · A06: Vulnerable & Outdated Components

Every category so far has been about *your* code or *your* configuration. **A06** is about everyone *else's*: the open-source libraries, frameworks, runtimes, and packages your application depends on. Modern apps are mostly other people's code — a typical project pulls in hundreds or thousands of dependencies — and a vulnerability in any one of them is a vulnerability in *your* app, even if your own code is flawless.

A06: Vulnerable & Outdated Components

This category covers using components (libraries, frameworks, servers, OS packages, container images) that are **out of date, unsupported, or have known vulnerabilities**. The defining feature: you didn't write the vulnerable code, but you *shipped* it. And because the vulnerable component is public and widely used, the exploit is public too — attackers scan for known-vulnerable versions at internet scale.

Why It's Uniquely Dangerous: Known & Weaponized

> **A KNOWN CVE IN A POPULAR COMPONENT IS A PUBLISHED EXPLOIT WAITING FOR A SCANNER**
>
> The hallmark of A06 is that the vulnerability is **publicly known** — it has a CVE, a description, often proof-of-concept exploit code, and a patch. That cuts both ways: defenders *could* fix it, but attackers have a ready-made exploit and a way to find every unpatched target. Automated scanners crawl the internet fingerprinting component versions (from headers, error pages, JS files, default paths) and fire known exploits at anything outdated — no skill required. So an unpatched component isn't a theoretical risk; it's a *countdown*. The window between a CVE's disclosure and mass exploitation is often days or hours.

The Canonical Example: Log4Shell

In December 2021, a critical vulnerability (**Log4Shell**, CVE-2021-44228) was found in **Log4j**, a ubiquitous Java logging library. A crafted string in anything that got *logged* (a username, a User-Agent header) triggered remote code execution. Because Log4j was buried as a transitive

dependency in a staggering number of applications and devices, *millions* of systems were instantly exploitable — many run by organizations that didn't even know they used Log4j. It's the perfect illustration of A06: a flaw in someone else's code, deep in the dependency tree, turning flawless applications into RCE targets overnight.

Transitive Dependencies: The Iceberg

The reason A06 is so pervasive is the **dependency tree**. You add one package; it depends on ten others, which depend on more, and so on. The components you *chose* (direct dependencies) are the tip; the vast majority are **transitive** — pulled in indirectly, often unknowingly.

```
# you installed 1 package — but your app actually ships dozens/hundreds
your-app
└─ web-framework        # direct (you chose it)
   ├─ template-engine   # transitive
   ├─ logging-lib  ←     # a Log4Shell-style flaw could live HERE, unseen
   └─ http-parser
      └─ ...deeper...    # you never explicitly added these
```

You can't patch what you don't know you have — which is why *visibility* into the full tree is the first defence.

The Defences

- **Inventory everything** — maintain a **Software Bill of Materials (SBOM)**: a complete list of all components and versions, direct and transitive. You can't secure what you can't see.
- **Scan continuously** — use **Software Composition Analysis (SCA)** tools (Dependabot, Snyk, OWASP Dependency-Check, `npm audit`, etc.) to flag dependencies with known CVEs, ideally in CI on every build.
- **Patch promptly** — keep components updated; treat critical-CVE patches as urgent. A documented, fast update process is the core control.
- **Remove what you don't use** — unused dependencies, features, and components are pure attack surface; prune them.
- **Use maintained, reputable sources** — prefer actively-maintained components from official repositories; avoid abandoned or unofficial packages.
- **Monitor advisories** — subscribe to security feeds (CVE/NVD, GitHub Security Advisories) for your stack so you hear about a Log4Shell on day zero, not day thirty.

A06 SHADES INTO SUPPLY-CHAIN SECURITY — THE COMPONENTS THEMSELVES CAN BE MALICIOUS

Beyond *outdated* components, the modern frontier is **supply-chain attacks**: the dependency isn't just vulnerable, it's *compromised*. Examples include **typosquatting** (a malicious package named like a popular one — `reqeusts` vs `requests`), **dependency confusion** (tricking a build into pulling an attacker's public package instead of your private one), and **compromised maintainer accounts** pushing a malicious update to a trusted package (the event-stream and SolarWinds incidents). Defences extend accordingly: pin exact versions and use **lockfiles**, verify integrity hashes, vet new dependencies before adding them, use private registries with namespacing, and — connecting to A08 — protect the build pipeline that pulls them in. The trust you place in a dependency is trust in everyone who can publish to it.

Hands-On Exercises

EXERCISE 1

Explain why a vulnerability in a third-party component is uniquely dangerous compared to a bug in your own code, focusing on "publicly known + weaponized + internet-scannable." Use Log4Shell to illustrate why transitive dependencies make it worse.

View solution

EXERCISE 2

Describe a practical A06 defence program: SBOM, SCA scanning in CI, patching process, removing unused deps, advisory monitoring. Explain what each step addresses and why "we'll update when we get around to it" fails given how fast known CVEs are exploited.

View solution

EXERCISE 3

Distinguish "vulnerable/outdated components" from "supply-chain attacks," with an example of each (a known CVE vs typosquatting/dependency confusion/compromised maintainer). List the additional defences supply-chain risk needs (lockfiles, integrity hashes, vetting, private registries) and how this connects to A08.

View solution

CHAPTER 6 QUICK REFERENCE

- **A06 Vulnerable & Outdated Components** — shipping libraries/frameworks/runtimes that are outdated, unsupported, or have known CVEs; you didn't write it, but you shipped it
- Uniquely dangerous: the flaw is **publicly known + weaponized + internet-scannable** — attackers fingerprint versions and fire ready-made exploits at scale
- **Log4Shell** (CVE-2021-44228) — a logged string → RCE in ubiquitous Log4j; millions exposed via a deep transitive dependency
- **Transitive dependencies** — most of your components are pulled in indirectly/unknowingly; you can't patch what you can't see
- Defences: **SBOM** (inventory all components) · **SCA scanning** in CI (Dependabot/Snyk/npm audit) · **patch promptly** · remove unused deps · use maintained sources · monitor advisories
- **Supply-chain** frontier: components can be *malicious* — typosquatting, dependency confusion, compromised maintainers; defend with lockfiles, integrity hashes, vetting, private registries
- Connects to **A08** (pipeline/build integrity) — protect what pulls the dependencies in
- **Next chapter:** A07 Identification & Authentication Failures (→ the Auth course)

OWASP Top 10

Course 1 · Chapter 7 · A07: Identification & Authentication Failures

This is a homecoming for anyone who did the **Authentication & Session Security** course — **A07** is that entire course's subject, compressed into one Top 10 slot. So this chapter's job is the *map*: what the category spans, why it ranks where it does, and where each piece links back to the depth you already have.

A07: Identification & Authentication Failures

This category covers failures in **confirming a user's identity** and **managing their session** securely — everything from weak passwords to broken session handling to flawed account recovery. Note the deliberate naming: it was previously "Broken Authentication," renamed in 2021 to also emphasize **identification** (establishing who someone claims to be) alongside authentication (proving it). It dropped from #2 to **#7** — not because it's less serious, but because frameworks and managed identity providers now handle a lot of it correctly out of the box.

What the Category Spans (and Where It's Covered)

FAILURE	EXAMPLE	AUTH-COURSE DEEP-DIVE
`Weak password storage`	plaintext / fast-hash passwords	Auth Ch. 2
`Credential brute force / stuffing`	no rate limit; reused breached passwords work	Auth Ch. 3
`Weak / no MFA`	password-only on high-value accounts	Auth Ch. 6
`Session management flaws`	predictable IDs, no rotation, fixation	Auth Ch. 4–5
`Exposed / mishandled tokens`	session/JWT in URLs, no invalidation	Auth Ch. 5, 7
`Weak account recovery`	security questions; guessable reset tokens	Auth Ch. 9
`Default / weak credentials`	`admin/admin` (also A05)	Auth Ch. 3

In other words: A07 ≈ the whole Auth course. Rather than re-teach it, this chapter pulls out the through-line that ties all those failures together.

The Unifying Idea: Protect One Secret Across Its Life

A07 IS "PROTECT THE CREDENTIAL, THEN THE SESSION THAT STANDS IN FOR IT"

The Auth course's organizing principle is the map for this whole category: authentication establishes an identity (a credential), then a **session** stands in for that credential across many requests — and the entire category is about protecting that one secret through its whole life. Store it so a breach can't reverse it (**hashing**), prove it without resending it (**sessions/tokens**), stop it being **guessed** (rate limits, MFA), **stolen** (HttpOnly, HTTPS), **fixated/hijacked** (regenerate at login, high-entropy IDs), or **bypassed** via recovery (secure reset tokens). Every A07 failure is a gap in one of those stages.

The Highest-Leverage Defences

If A07 is the Auth course condensed, here are the controls that matter most — in rough order of impact:

1. **Multi-factor authentication** — the single highest-leverage control; a guessed or stuffed password alone is no longer enough (Auth Ch. 6).
2. **Strong password storage** — slow, salted hashes (Argon2id/bcrypt), never fast hashes or "encryption" (Auth Ch. 2; also A02).
3. **Defend the login** — rate limiting (IP + account), breach-password checks, generic responses + uniform timing (no enumeration) (Auth Ch. 3).
4. **Secure sessions** — high-entropy IDs, regenerate at login (anti-fixation), idle + absolute timeouts, server-side logout, `HttpOnly` / `Secure` / `SameSite` cookies (Auth Ch. 4–5).
5. **Secure recovery** — single-use, expiring, hashed reset tokens; re-authenticate sensitive changes; no security questions (Auth Ch. 9).
6. **Don't roll your own** — use vetted auth libraries / managed identity providers (which is largely *why* A07's rank fell).

AUTHENTICATION IS ONLY AS STRONG AS ITS WEAKEST PATH — INCLUDING RECOVERY

The recurring Auth-course lesson, stated at Top 10 level: an attacker attacks the *weakest* way in, not the strongest. Passkeys and MFA on the front door mean nothing if account recovery falls back to an

SMS code or a security question (the **#1 takeover vector**). Likewise, perfect password hashing doesn't help if the session ID can be guessed, and strong sessions don't help if XSS can steal them (which is why "fix XSS first" recurs). A07 is a system property: every stage — storage, login, session, MFA, recovery — must hold, because account security is set by the weakest link across all of them, and it interlocks with A01 (authorization), A02 (crypto), A03 (XSS stealing sessions), and A05 (default creds).

Identification vs Authentication — the Rename's Point

The 2021 addition of "Identification" highlights a subtle layer: **identification** is the *claim* ("I'm user philip"), **authentication** is the *proof* (the password/passkey checks out). Failures can occur at the identification layer too — e.g. allowing account enumeration (confirming which identities exist), weak identity proofing during registration, or letting one identity be confused with another. Getting identity right is the precondition for getting authentication right.

Hands-On Exercises

EXERCISE 1

A07 spans many failures. For each, name the stage of the credential/session lifecycle it breaks and the primary defence: (a) plaintext passwords; (b) no login rate limit; (c) the same session ID after login; (d) password reset via security questions. Then state the one unifying idea behind all of them.

View solution

EXERCISE 2

Explain why "authentication is only as strong as its weakest path," using the recovery flow as the example. Then show how A07 interlocks with A01, A02, A03, and A05 — give one concrete way each of those categories can undermine authentication.

View solution

EXERCISE 3

Explain why A07 dropped from #2 to #7 and what the "Identification" rename adds. Then prioritize the top defences (MFA, password storage, login defence, sessions, recovery, don't-roll-your-own) and

justify why MFA tops the list.

📄 View solution

CHAPTER 7 QUICK REFERENCE

- **A07 Identification & Authentication Failures** — failures confirming identity & managing sessions; ≈ the whole Auth course; renamed from "Broken Authentication," dropped #2 → #7 (frameworks/IdPs help)
- Spans: weak password storage · brute force/stuffing · weak/no MFA · session flaws (predictable IDs, fixation) · exposed tokens · weak recovery · default creds
- **Unifying idea**: protect one secret (credential → session) across its life — store (hash) · prove (session) · don't let it be guessed/stolen/hijacked/bypassed
- Top defences: **MFA** (highest leverage) · slow salted hashing · login rate-limit + no enumeration · secure sessions (regenerate at login, cookie flags, timeouts) · secure recovery · don't roll your own
- **Weakest-path rule**: recovery is the #1 takeover vector — strong login + weak reset = weak; XSS steals sessions ("fix XSS first")
- Interlocks with **A01** (authz), **A02** (crypto/hashing), **A03** (XSS session theft), **A05** (default creds)
- Deep-dive: the entire **Authentication & Session Security** course
- **Next chapter:** A08 Software & Data Integrity Failures — deserialization, unsigned updates, CI/CD integrity

OWASP Top 10

Course 1 · Chapter 8 · A08: Software & Data Integrity Failures

A08 is a relatively new category (added in 2021) that consolidates a critical idea: you must be able to **verify that code and data haven't been tampered with** — whether at rest, in transit, or during the build/deploy pipeline. It's the inverse of confidentiality: instead of "nobody can read this," it's "I can prove nobody *changed* this." This chapter covers the three biggest threat surfaces: **deserialization**, **CI/CD pipeline integrity**, and **insecure software updates**.

A08: Software & Data Integrity Failures

This category spans assumptions about code/data being authentic and unmodified — and the attacks that exploit when those assumptions break. It's fundamentally about **trust**: can you prove the code you're running is the code that was built? Can you prove the data came from where you think? Without integrity checks, every layer of security collapses — an attacker who can modify code or data owns the system, full stop.

Deserialization Attacks (the "Untrusted Input" Extreme)

Serialization is the process of turning an object (a class instance, a data structure) into a byte string so you can store or transmit it. **Deserialization** is reconstructing the object from those bytes. Many languages provide "native" serializers (Java's `ObjectInputStream`, Python's `pickle`, PHP's `unserialize`) that are **expressive but dangerous**: they can reconstruct objects by executing methods during deserialization, which means a crafted byte string can trigger arbitrary code.

DESERIALIZATION: CODE EXECUTION VIA OBJECT RECONSTRUCTION

If an attacker can control the byte string being deserialized, they can craft one that, when deserialized, instantiates a class and calls methods that execute code (e.g. opening a file, running a command, starting a reverse shell). This is why deserializing untrusted input (user uploads, network messages, database blobs) with a language's native serializer is an **RCE vulnerability** — as bad as SQL injection or command injection. The fix: **never deserialize untrusted data**. Use **safe formats** (JSON, Protocol Buffers, msgpack) that only produce data, not executable objects, and validate the structure after parsing.

CI/CD Pipeline Integrity (the Build Supply Chain)

The **CI/CD pipeline** is your automated build-and-deploy system — and it is a prime attack surface. An attacker who gains access to the pipeline can inject malicious code into the build, sign it with your credentials, and ship it to production without anyone noticing (unless you're watching). A08 defences here are:

- **Least privilege** — CI/CD workers (containers, VMs) run with minimal permissions; they can't ssh to prod, can't access secrets they don't need, can't push to main.
- **Audit logging** — every CI/CD action (build start, secret access, deploy) is logged and monitored; unusual patterns trigger alerts.
- **Code review before merge** — changes to production code require human review and approval; no auto-merge from a build or a bot (or if it exists, it's narrowly scoped and logged).
- **Artifact signing** — the build produces a binary/image/package with a cryptographic signature; the deploy step verifies the signature before running it.
- **Signed commits** — code commits are GPG-signed; the CI system verifies the signature and fails unsigned/invalid pushes (prevents an attacker who gains repo access from committing unsigned code).

The principle: **every step in the pipeline must be auditable, and code/artifacts must be signed so you can prove they came from an authorized person/system.**

Insecure Software Updates (Trusting Without Verifying)

Many apps auto-update without user intervention or verification. If the update mechanism doesn't verify the authenticity and integrity of the update, an attacker who can intercept or replace the update (via DNS hijacking, MITM, or compromised CDN) can push malicious code directly to users. A08 defences:

- **Sign updates** — the update (binary, package, script) is cryptographically signed by the vendor's private key.
- **Verify the signature before installing** — the client checks the signature using the vendor's public key; unsigned or invalid updates are rejected.

- **Use HTTPS for downloads** — prevents simple interception (though doesn't stop a compromised server).
- **Staged rollouts** — don't push the same version to all users at once; watch for crashes/anomalies in early adopters before rolling out globally.

The Integrity Idea: Trust is Proof, Not Faith

> **A08 = "VERIFY, DON'T ASSUME"**
>
> A08 collapses into one idea: **never assume data or code is authentic or unmodified just because it came from a familiar source or format.** Verify it cryptographically. Hashing (SHA-256) proves *tampering* (any change breaks the hash). Signing (asymmetric crypto) proves *authorship* (only the holder of the private key could have signed it). Encryption proves *confidentiality* (only the holder of the key can read it) — but *not* integrity; use authenticated encryption (AES-GCM) or sign + encrypt. The three places this matters most: (1) **deserialization** — only deserialize data you've authenticated; (2) **CI/CD** — sign commits, artifacts, deployments, and log everything; (3) **software updates** — sign updates and verify before installing.

Interlocks and Connections

A08 doesn't stand alone:

- **A06 (Vulnerable & Outdated Components)** — A06 is about getting the component versions right; A08 is about ensuring the supply chain that delivers them is trustworthy. Together: inventory your deps (A06) + verify the build/deploy pipeline (A08) = you can ship with confidence.
- **A02 (Cryptographic Failures)** — A08 *uses* cryptography (signing, hashing, authenticated encryption) to enforce integrity; weak crypto breaks A08's defences.
- **A05 (Security Misconfiguration)** — not signing updates, not verifying signatures, auto-merge without review, overprivileged CI workers — these are all misconfigurations that enable A08 attacks.
- **A03 (Injection)** — if an attacker injects malicious code into your code repository (via a compromised package, a MITM, or a CI/CD flaw), it is then built and shipped as part of A08's supply-chain attack.

Hands-On Exercises

EXERCISE 1

Explain why deserializing untrusted data with Java's `ObjectInputStream` (or Python's `pickle`) is an RCE vulnerability. Then contrast this with deserializing JSON; explain why JSON is safe and how you should validate it.

View solution

EXERCISE 2

Design a CI/CD pipeline's integrity controls. List 5 defences (from least privilege to signed artifacts) and for each, explain: (a) what it prevents, (b) how an attacker exploits its absence. Then explain why code review before merge is the most critical one.

View solution

EXERCISE 3

Explain the difference between hashing, signing, and authenticated encryption in terms of what each proves (tampering, authorship, confidentiality). Then give a concrete attack on an auto-update system that has no verification and explain how each of the three would stop it.

View solution

CHAPTER 8 QUICK REFERENCE

- **A08 Software & Data Integrity Failures** — failures to verify code/data is authentic & unmodified; proves trust, not assumes it; new in 2021
- **Deserialization RCE**: native serializers (ObjectInputStream, pickle) execute methods during deserialization; crafted input = code execution. Fix: never deserialize untrusted data; use safe formats (JSON, protobuf) + validate.
- **CI/CD Pipeline Integrity**: least-privilege workers · audit logging · code review before merge · signed commits · signed artifacts + verification. An attacker in the pipeline ships malicious code; every step must be auditable & signed.

- **Insecure Software Updates**: auto-update without verification = attacker can push malicious code. Fix: sign updates + verify signature before install; use HTTPS; staged rollouts.
- **Integrity idea**: **verify, don't assume.** Hashing proves tampering. Signing proves authorship. Authenticated encryption proves confidentiality + integrity. Use all three in the supply chain.
- Interlocks with **A06** (dependencies + supply chain), **A02** (crypto), **A05** (misconfig), **A03** (code injection)
- **Next chapter:** A09 Security Logging & Monitoring Failures — detecting attacks after they happen

OWASP Top 10

Course 1 · Chapter 9 · A09: Security Logging & Monitoring Failures

All the prior categories (A01–A08) are about **preventing specific attacks**. **A09** is different: it's about **visibility**. Even if you prevent 99% of attacks perfectly, the 1% that gets through is worthless to know about unless you can **see it happening** and **respond to it**. A09 is the category that answers: "Do you have logs? Are they trustworthy? Are you watching them? Do you know when you've been breached?"

A09: Security Logging & Monitoring Failures

This category covers the absence or inadequacy of security event logging and real-time monitoring — failures that mean attacks go undetected, breaches persist for months, and forensics after the fact are impossible. The core failures are:

- **Insufficient logging** — not capturing security-relevant events (auth attempts, access control failures, input validation failures, suspicious patterns).
- **Logs not protected** — logs are stored in the application server (where an attacker can delete them) or sent unencrypted (where they can be intercepted or modified).
- **No monitoring** — logs exist but nobody is reading them; no alerts on suspicious activity; no incident response plan.
- **Poor log visibility** — logs are too verbose or too sparse; signal is buried in noise, or critical events are absent.

What to Log (and What NOT to Log)

DO LOG (security events):

- **Authentication** — login attempts (success and failure), MFA challenges, session creation/destruction, password resets.
- **Authorization** — access control decisions (allowed and denied), privilege escalation attempts, attempts to access resources outside the user's scope.
- **Input validation** — validation failures (inputs that don't match expected patterns), suspicious payloads, repeated failed validations from the same IP.

- **Data access** — who accessed what data and when (especially sensitive data).
- **Configuration changes** — admin actions, deployment events, security settings changes.
- **Errors** — unexpected errors, especially security-related (crypto failures, deserialization errors, CORS violations).

DO NOT LOG (sensitive data):

- Passwords, API keys, tokens, secrets (even in error messages).
- Personally identifiable information (PII): credit card numbers, SSNs, email addresses, phone numbers.
- Health data, financial data, or other regulated information unless required by compliance and redacted.

> **GOOD LOGGING IS A SECURITY CONTROL AND A COMPLIANCE REQUIREMENT**
>
> Logging is two things: a security control (detecting attacks) and a compliance requirement (HIPAA, PCI DSS, GDPR, SOC 2). But logging sensitive data in plaintext is itself a breach vector — an attacker who gains access to logs obtains credentials. So redact passwords, tokens, and regulated PII, but log the fact that they were provided (e.g. "password reset request from user X" without logging the new password).

Log Integrity: Protecting the Logs

Logs are a crime scene. If an attacker can modify or delete them, they can cover their tracks. A09 defences for log integrity:

- **Centralized logging** — send logs to a separate system (a logging service, a SIEM, a write-only database) that the application can't delete. If logs stay on the app server, an attacker who compromises the app deletes them.
- **Write-once storage** — logs are appended to immutable storage (e.g. a write-once tape, or a cloud service with audit logs that can't be deleted, like AWS CloudTrail). Once written, they can't be modified or deleted.
- **Signed/hashed logs** — each log entry is signed or chained (e.g. each entry includes a hash of the previous entry). Tampering breaks the chain and is detectable.
- **Encryption in transit** — logs are sent over HTTPS/TLS; they're not readable if intercepted.

- **Access control** — only authorized personnel can read logs; an attacker with app-level access doesn't have log access.

Monitoring and Alerting (the Human/Automated Response)

Logs are useless if nobody reads them. A09 defences for visibility:

- **Real-time monitoring** — a SIEM (Security Information & Event Management) tool watches logs as they arrive and flags anomalies (e.g. 10 failed logins from the same IP, a user accessing 1000 files in 1 second, a privilege escalation).
- **Alerting** — automated alerts to on-call security staff when suspicious activity is detected (email, SMS, PagerDuty).
- **Baselines and thresholds** — know what "normal" looks like (typical login times, typical data access patterns) so you can spot when activity deviates.
- **Incident response plan** — when an alert fires, who is on-call? What's the playbook? Can you kill a compromised session, revoke a stolen token, isolate a server?

> **LOGS ALONE DON'T HELP; DETECTION REQUIRES SOMEONE ON THE OTHER END**
>
> A frequent failure: comprehensive logging exists, but nobody monitors the logs. Attackers operate in the logs for weeks before discovery happens (if at all). "We knew about it the whole time, it was in the logs" is a post-breach regret. Prevention: alert on unusual activity (both automated rules and human spot-checks of logs during on-call shifts), and have an incident response plan so alerts are actionable.

Breach Detection: How A09 Connects to Everything Else

A09 doesn't prevent any of the attacks in A01–A08. But it lets you **detect** them:

- **A01 (Access Control)** — log access control failures; alert on repeated denials or escalation attempts.
- **A02 (Crypto)** — log crypto errors; they're usually rare and often indicate an attack or misconfiguration.
- **A03 (Injection)** — log input validation failures; SQL errors that indicate injection attempts.
- **A04 (Insecure Design)** — log business-logic anomalies (large transactions, unusual access patterns, price manipulations).

- **A05 (Misconfiguration)** — log config changes; alert on changes to security settings.
- **A06 (Outdated Components)** — log library version changes and known-CVE scanners; alert on vulnerable versions reaching production.
- **A07 (Auth)** — log auth events extensively (login attempts, MFA, sessions, recovery).
- **A08 (Integrity)** — log artifact deployments, signed-commit validations, CI/CD actions.

A09 is the **safety net** — it doesn't prevent breaches, but it catches them before they become catastrophic.

Hands-On Exercises

EXERCISE 1

Design a logging strategy for an e-commerce app. List 5 security events to log (with example data), and for each event, identify what sensitive data should be redacted to avoid exposing secrets/PII. Then explain why centralizing logs to a separate service is critical.

View solution

EXERCISE 2

Explain why monitoring without alerting is useless. Then design an alerting strategy: list 5 alerts (with thresholds) that would detect ongoing attacks (e.g. brute force, data exfiltration, privilege escalation). For each, explain what response action the on-call team should take.

View solution

EXERCISE 3

Explain how log tampering defeats breach detection and why centralized, write-once logging is critical. Then give a concrete attack (attacker compromises the app server) and show how insufficient logging vs. proper logging would change the outcome.

View solution

CHAPTER 9 QUICK REFERENCE

- **A09 Security Logging & Monitoring Failures** — absence/inadequacy of event logging & real-time monitoring; attacks go undetected; breaches persist for months
- **What to log:** auth (login, MFA, session) · authz (access decisions, denials) · input validation failures · data access · config changes · errors (especially security-related)
- **What NOT to log:** passwords, API keys, tokens, secrets · PII (credit cards, SSNs, email, phone) · health/financial data (unless redacted per compliance)
- **Log integrity:** centralize to separate system (attacker can't delete) · write-once storage (immutable) · signed/hashed chains (detect tampering) · encrypt in transit · restrict access
- **Monitoring & alerting:** real-time SIEM watching for anomalies · automated alerts on suspicious activity · baselines & thresholds (know what normal looks like) · incident response plan (on-call team knows what to do)
- **A09 connects to all others:** doesn't prevent attacks, but detects them when they slip through
- **Key insight:** logs are a crime scene; tampered/absent logs mean breach detection fails and attackers cover their tracks
- **Next chapter:** A10 Server-Side Request Forgery (SSRF)

OWASP Top 10

Course 1 · Chapter 10 · A10: Server-Side Request Forgery

A10 is about **forging requests on the server's behalf**. If a web server fetches a URL provided by an attacker, the attacker can make the server request internal resources, cloud metadata, or other services — effectively using the server as a proxy into the network the server trusts. SSRF is the server-side cousin of CSRF (which forges requests from the browser). The CSRF course touched on this as a related attack; A10 brings it into focus.

A10: Server-Side Request Forgery (SSRF)

This category covers vulnerabilities where the server makes HTTP requests to a URL that an attacker controls (or influences). The attacker crafts a URL pointing to an internal resource, the server fetches it, and the attacker learns something (or causes an action) they shouldn't be able to. Common targets are **internal APIs, cloud metadata services, localhost services, databases**, and **internal dashboards**.

The Attack Landscape (What Attackers Target)

TARGET	WHAT IT IS	EXAMPLE PAYLOAD	WHY IT'S DANGEROUS
AWS Metadata Service	Cloud metadata endpoint (169.254.169.254)	`http://169.254.169.254/latest/meta-data/iam/security-credentials/role-name`	Exposes AWS credentials, role tokens, instance details
Localhost Services	Internal services on port 80, 8080, 5432, 27017	`http://localhost:27017` (MongoDB)	Access to DBs, admin dashboards, not intended to be public
Internal APIs	APIs only on internal network (e.g., payment processor, CRM)	`http://internal-api.corp/admin/users`	Attacker makes requests to APIs they shouldn't access

TARGET	WHAT IT IS	EXAMPLE PAYLOAD	WHY IT'S DANGEROUS
Private IPs	Addresses in 10.x, 172.16–31.x, 192.168.x	`http://10.0.0.5/admin`	Enumeration of internal network, discovery of services
File URLs	Local filesystem access (file://)	`file:///etc/passwd`	Read sensitive files on the server
Gopher/Dict	Legacy protocols (Gopher, Dict)	`gopher://localhost:25` (SMTP)	Access to ports/services not intended for HTTP, bypass filters

How SSRF Attacks Work

The attack flow:

1. **Attacker-controlled URL parameter** — app has a feature like "fetch and embed a preview of this URL" or "download a file from this URL" or "webhook callback to this endpoint."
2. **Attacker crafts a malicious URL** — pointing to an internal resource (e.g., `http://localhost:8080/admin`, or `http://169.254.169.254/latest/meta-data/`).
3. **Server makes the request** — the server's HTTP client (curl, requests, urllib) fetches the URL on the attacker's behalf.
4. **Attacker gains information or causes an action** — either the response is returned to the attacker (informative SSRF), or the attacker infers success from side effects (blind SSRF, timing, error messages).

SSRF IS OFTEN SILENT

Unlike SQL injection (which produces errors), SSRF often succeeds silently. The server fetches the internal URL, and the attacker sees either: (a) the response content (metadata, data, admin pages), (b) a timing difference (service is up vs. down), or (c) an error message (e.g., "MongoDB connection failed" leaks that a MongoDB is running on localhost). Blind SSRF (where the attacker learns nothing directly) can still be chained with other attacks (exfiltration via DNS requests, exploitation of internal services).

SSRF vs CSRF (Cousins, Not Twins)

Both are forgery attacks, but the actor is different:

- **CSRF (Cross-Site Request Forgery)** — attacker tricks the **user's browser** into making a request (e.g., POST to /transfer-money). The request comes from the user's IP and session. The user's browser makes the request.
- **SSRF (Server-Side Request Forgery)** — attacker tricks the **server** into making a request (e.g., fetch a URL). The request comes from the server's IP and network privileges. The server makes the request.

CSRF is blocked by same-site cookies, CSRF tokens, and origin checks. SSRF is blocked by URL validation and network filtering (don't allow the server to request internal resources).

Defences Against SSRF

1. URL Validation / Allowlisting (the primary defence)

- **Allowlist expected domains** — if the feature is "fetch an image from a user-provided URL," explicitly list which domains are allowed (e.g., only `images.cdn.example.com`). Reject anything else.
- **Block dangerous IP ranges** — 127.0.0.1, localhost, 0.0.0.0, 169.254.169.254 (AWS), 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 (private networks). Reject any URL that resolves to these.
- **Validate after DNS resolution** — an attacker might use DNS rebinding (first lookup returns attacker's IP, second returns localhost). Validate the resolved IP, not the hostname.

2. Network Filtering (defence in depth)

- **Egress filtering** — the server's firewall rules prevent outbound connections to internal IPs or certain ports (e.g., database ports 5432, 27017, 3306). The server can't reach them, even if SSRF is exploited.
- **Restricted HTTP client** — use an HTTP client that doesn't follow redirects (an attacker could redirect to an internal URL), doesn't support dangerous schemes (file://, gopher://), and enforces timeouts.

3. Least Privilege for Outbound Requests

- **Separate network segment** — outbound requests come from a sandboxed service with no access to internal APIs / databases / metadata.
- **VPC/network isolation** — the server is in a subnet that cannot reach internal services (only specific, whitelisted external services).

Hands-On Exercises

EXERCISE 1

Explain SSRF in contrast to CSRF: who makes the request, from where, and what does an attacker gain? Then give 3 SSRF attack targets (AWS metadata, localhost DB, internal API) and explain what each exposes.

View solution

EXERCISE 2

Design a URL validation defence against SSRF. List 5 dangerous IP ranges / hostnames to block, explain why each is dangerous, and then explain why blocking by hostname alone (without DNS resolution) is insufficient (DNS rebinding attack).

View solution

EXERCISE 3

Sketch a feature that's vulnerable to SSRF ("fetch a preview from a URL"). Then design the code with 3 defences: (a) URL validation/allowlisting, (b) network filtering, (c) least privilege. Explain how each defence independently prevents the attack.

View solution

CHAPTER 10 QUICK REFERENCE

- **A10 Server-Side Request Forgery** — server makes HTTP requests to attacker-controlled URL; attacker targets internal resources (APIs, metadata, DBs, localhost)

- **Common targets:** AWS metadata (169.254.169.254) · localhost services (ports 5432, 27017, 8080) · internal APIs · private IPs (10.x, 172.16–31.x, 192.168.x) · file:// URLs
- **SSRF vs CSRF:** CSRF forges browser requests; SSRF forges server requests. SSRF comes from server's IP/network privilege, reaching internal resources.
- **Defences:** URL validation + allowlisting (block private IPs/ranges, localhost, metadata services) · validate AFTER DNS resolution (detect DNS rebinding) · network egress filtering (prevent server from reaching internal ports) · restricted HTTP client (no redirects, no dangerous schemes) · least privilege (server can't reach internal services)
- **SSRF is often blind:** attacker learns from timing, error messages, or side effects; not always obvious like SQLi
- **Next:** Combined OWASP Top 10 summary & how the 10 categories interconnect

OWASP TOP 10 (2021) — THE FULL PICTURE

A01 Access Control — who can do what; authz checks, IDORs, forced browsing

A02 Crypto Failures — weak/missing encryption, bad key mgmt, weak hashing

A03 Injection — SQL, OS command, SSTI, XSS (absorbed); data vs. code

A04 Insecure Design — threat modeling, secure-by-design, missing defences before code

A05 Misconfiguration — default creds, verbose errors, missing security headers, IaC drift

A06 Outdated Components — known-vulnerable deps, supply chain, SBOM/SCA/patching

A07 Auth Failures — weak passwords, no MFA, session flaws, weak recovery

A08 Integrity Failures — deserialization RCE, CI/CD pipeline, unsigned updates, artifact signing

A09 Logging Failures — insufficient logs, tampered logs, no monitoring/alerting

A10 SSRF — server forges requests to internal resources; URL validation & network filtering

Interconnections: A01 (authz) + A07 (auth) = system identity. A02 (crypto) + A08 (integrity) = trust verification. A03 (injection) + A04 (design) = preventing hostile input. A05 (miscfg) + A06 (deps) + A09 (logging) = operational security. A10 (SSRF) + A01 (authz) = internal attack surface. All 10 depend on threat modeling (A04) and visibility (A09).