OWASP Top 10

Version 2025 · verified 2026-08-10

The ten 2025 categories, what changed since 2021, and the control that actually prevents each one.

Contents

Mental model

The Top 10 is an awareness document, not a standard and not a checklist you can pass. It ranks **root causes** by how widely they appear across the industry, which is why 2025 folded SSRF into Broken Access Control and grew “Vulnerable and Outdated Components” into Software Supply Chain Failures. Position on the list is prevalence everywhere, not risk in your application — and a category you have never triaged is more dangerous than a high-ranked one you already control.

Reading the 2025 list

Input to the ranking	2025 figure
CWEs analysed	589, of which 248 rank
Applications tested	2.8 million, 13 organizations
Categories drawn from data	8

Input to the ranking	2025 figure
Categories drawn from survey	2

Ranking uses **incidence rate** — the share of tested applications with at least one instance of a CWE — so an app with 5,000 findings counts once and noisy scanner categories cannot dominate. Exploitability and impact weights come from roughly 175,000 CVE records mapped to CWEs. Only eight slots are data-driven, because data reports what tools already have rules for; the remaining two come from the community survey to cover risk the scanners cannot see yet.

```
A01 Broken Access Control
A02 Security Misconfiguration
A03 Software Supply Chain Failures
A04 Cryptographic Failures
A05 Injection
A06 Insecure Design
```

```
A07 Authentication Failures
A08 Software or Data Integrity Failures
A09 Security Logging and Alerting Failures
A10 Mishandling of Exceptional Conditions
```

> **Gotcha:** The `A0n` number is a position, not an identity. `A03` was Injection in 2021 and is Software Supply Chain Failures in 2025, so a ticket or policy citing a bare `A03` is ambiguous. Always write the year.

Broken access control (A01)

Failure	What it looks like
No check on state-changing verbs	`POST`, `PUT`, `DELETE` open
Insecure direct object reference	`?acct=` someone else's id
Enforcement in the client only	`curl` calls the endpoint

Failure	What it looks like
Metadata tampering	JWT or cookie edited to admin
SSRF, merged in for 2025	Server fetches an attacker URL

Access control enforces policy so that users cannot act outside their intended permissions, and it stays at number one: 40 CWEs, present in up to 20.15% of tested applications. Deny by default for everything non-public, enforce server-side through one reused mechanism, and check the **record**, not just the route — ownership is what most route guards forget. SSRF joined this category because it is the same failure seen from the server's side.

```
// Route guard only: any logged-in user
// can read any order.
app.get("/orders/:id", auth, (req, res) =>
  res.json(db.order(req.params.id)));
```

```
// Object-level check: the record has to
// belong to the caller.
app.get("/orders/:id", auth, (req, res) => {
  const o = db.order(req.params.id);
  if (o?.userId !== req.user.id)
    return res.sendStatus(404);
  res.json(o);
});
```

> **Gotcha:** Hiding the admin button is not access control. Anything the browser enforces is advisory — the endpoint is one `curl` away, and the attacker never loads your JavaScript.

Security misconfiguration (A02)

Misconfiguration	Result
Default credentials kept	Direct admin access

Misconfiguration	Result
Verbose errors to the client	Reconnaissance material
Directory listing enabled	Source and class download
Cloud storage open by default	Public data exposure
Security headers missing	Client-side defences off
XML external entities enabled	File read and SSRF (CWE-611)

Second place, up from fifth, and holder of the highest single incidence rate in the dataset at 27.70%. This is a process failure, not a code failure: the fix is one automated hardening path every environment goes through, over a minimal platform with sample apps and unused features removed. Prefer identity federation and short-lived credentials over static secrets, and keep a central error handler so nothing leaks a stack trace by accident.

```
// NODE_ENV unset means development mode,
// which returns stack traces to the client.
app.use((err, _req, res, _next) => {
  logger.error(err);            // detail stays
  res.status(500).json({ error: "Internal" });
});
```

> **Warning:** An upgrade can quietly restore a vendor default you had turned off. Re-run the configuration check after every version bump, in every environment — staging drifting from production is how most of these reach users.

Supply chain and integrity (A03, A08)

Layer	Compromise
Direct dependency	Known CVE left unpatched
Transitive dependency	A package you never chose
Install script	Credential theft during install

Layer	Compromise
Build system or CI/CD	Signed artifact, injected code
Update or plugin (A08)	Unsigned firmware replaced
Serialized input (A08)	Deserialization to RCE

A03 is new at number three, expanded from A06:2021 to cover the whole ecosystem that produces your artifact — half of survey respondents ranked it first, and it carries the highest average incidence rate at 5.72%. SolarWinds reached roughly 18,000 organizations through a signed vendor update, and 2025's Shai-Hulud npm worm self-propagated across 500+ package versions using stolen tokens. A08 is the same trust question one level down at runtime: did you verify the update, plugin, or serialized object before executing it? Keep an SBOM of direct and transitive dependencies, pull only from official sources, and sign your builds.

```
# Install without running package scripts.
npm ci --ignore-scripts

# Inventory what actually shipped.
npm sbom --sbom-format cyclonedx > sbom.json
```

> **Gotcha:** A lockfile pins versions, not behaviour. Install scripts still execute arbitrary code with your shell's privileges and your registry tokens, which is exactly how Shai-Hulud spread from one developer machine to the next package.

Cryptographic failures (A04)

Failure	Replacement
Cleartext transport	TLS 1.2+ with HSTS
Fast hash for passwords	Argon2, scrypt, PBKDF2
`Math.random()` for tokens	The OS CSPRNG
Hard-coded key in the repo	A KMS or HSM

Failure	Replacement
Encryption without integrity	An authenticated mode

Thirty-two CWEs, and weak pseudo-random number generation accounts for three of the most frequent. Classify what needs protection first — data under GDPR or PCI DSS, credentials, health records — then encrypt it in transit and at rest using established implementations, never your own construction. Use authenticated encryption, never reuse an IV, and plan for post-quantum algorithms landing by 2030.

```
import { randomBytes } from "node:crypto";

// Predictable: Math.random is not a CSPRNG,
// and its output leaks its own state.
const weak = Math.random().toString(36);
```

```
// Unguessable: 256 bits from the OS.
const token = randomBytes(32).toString("hex");
```

> **Gotcha:** SHA-256 is a fine hash and a terrible password hash — speed is the attacker's advantage on a GPU. Password storage needs a deliberately slow, salted KDF: Argon2, scrypt, or PBKDF2-HMAC-SHA-512.

Injection (A05)

Interpreter	CWEs
SQL and ORM query	CWE-89, CWE-564
Browser DOM (XSS)	CWE-79, CWE-80
OS command	CWE-77, CWE-78
LDAP filter	CWE-90
Expression language	CWE-917

Injection covers 37 CWEs and more CVEs than any other category — 62,445 — and since 2021 it has included cross-

site scripting, because an HTML page is just another interpreter. The concept is identical everywhere: data must never be able to become structure. Parameterize the query, or use an API that has no interpreter at all; positive server-side validation is a useful second line but not a substitute.

```
// Input becomes query structure.
db.query(
  `SELECT * FROM users WHERE id = '${id}'`);

// Parameterized: input can only be a value.
db.query(
  "SELECT * FROM users WHERE id = $1", [id]);
```

> **Gotcha:** Table and column names cannot be parameterized, and OWASP notes they cannot be escaped either. Any sort-by-column or report builder taking a name from the user needs an allowlist of permitted identifiers — nothing else works.

Insecure design (A06)

Design gap	How it shows up
No threat model	Control missing, not broken
Unbounded business flow	500 seats held, none bought
No bot or rate control	Scalpers clear the stock
Knowledge-based recovery	Reset by public trivia

A secure design can still have implementation defects, but a flawed design cannot be rescued by flawless code

— the control was never there to get wrong. That makes this a lifecycle category: threat-model the critical flows (authentication, access control, business logic), write security requirements into user stories, and segregate tenants by design. Test misuse cases alongside the happy path — that is where an unbounded flow shows itself.

```
declare function hold(
  req: { seats: number },
): Promise<{ status: number }>;

// Test the abuse case, not just the flow.
const res = await hold({ seats: 500 });
if (res.status !== 400) {
  throw new Error("seat hold is uncapped");
}
```

> **Gotcha:** A pen test finds implementation bugs; it rarely finds a control nobody specified. Nothing in a scanner reports “this booking flow has no limit” — only a threat model does.

Authentication failures (A07)

Weakness	Countermeasure
Credential stuffing	MFA, breached-password checks
Password spraying	Rate limit, failed-login alerts
Default or hard-coded creds	Never ship them (CWE-798)
Session fixation	New random ID after login
Session ID in the URL	Cookie, `Secure`, `HttpOnly`
SSO without single logout	Invalidate every linked session

Renamed from “Identification and Authentication Failures” to match the 36 CWEs it now spans, holding at number seven. Follow NIST 800-63B section 5.1.1: screen new passwords against breached and worst-password lists, and drop arbitrary rotation. Sessions matter as much as login — use the platform’s session manager, issue a fresh high-entropy ID after any privilege change, and invalidate server-side on logout and timeout. Validate the JWT `aud` and `iss` claims and scopes.

```
// Session fixation: reusing the pre-login ID
// lets whoever planted it ride in with you.
req.session.regenerate(() => {
  req.session.userId = user.id;
});
```

> **Gotcha:** Forced 90-day rotation is against current NIST guidance and actively harmful — it manufactures `Winter2025` becoming `Winter2026`, the exact pattern hybrid credential-stuffing tools try first. Rotate on evidence of compromise, not on a calendar.

Detection and failure paths (A09, A10)

Failure	What it costs
Auth and access failures unlogged	Nothing to detect with
Logs written, no alert threshold	Third party reports your breach
Stack trace returned to the user	Free reconnaissance
Partial rollback after an error	Corrupt state, drained account
Resource not released in a handler	Exhaustion, denial of service

A09 gained “Alerting” in its name for 2025 because logs nobody is paged for are not a control: one OWASP scenario ran seven years undetected across 3.5 million health records, another ended in a £20 million GDPR fine. A10 is the other new category — failing to prevent, detect, or sensibly respond to abnormal conditions, covering fail-open logic (CWE-636), leaked error detail (CWE-209), and null dereferences (CWE-476). Handle errors where they occur, roll the whole transaction back rather than half-recovering, and keep a global handler beneath that as the net.

```
try {
  await charge(order);
} catch (err) {
  await tx.rollback();          // fail closed
  log.warn({ orderId, code: err.code });
  res.status(502).json({ error: "Retry" });
}
```

> **Gotcha:** A log is a sink like any other. Unescaped newlines in logged input let an attacker forge entries (CWE-117), and dumping a whole request body writes credentials and tokens into a store with far weaker access control than your database (CWE-532).

Further reading

- OWASP Top 10:2025
- Methodology and data
- A01 Broken Access Control
- A03 Software Supply Chain Failures
- A10 Mishandling of Exceptional Conditions
- OWASP Cheat Sheet Series