ChatGPT

What is OWASP Top 10? A Beginner-Friendly Guide

The **Open Web Application Security Project (OWASP)** is a community-driven nonprofit dedicated to improving software security [1]. Its most famous resource, the **OWASP Top 10**, is a list of the most critical web application security risks, determined by a broad consensus of experts [2] [3]. It serves as a simple awareness tool – a go-to checklist of the biggest issues to avoid when building or using web applications.

The OWASP Top 10 is updated every few years to reflect new threats. The latest version (2021) added categories like **Insecure Design** and **Server-Side Request Forgery (SSRF)** and reshuffled others. Since many websites share similar components and frameworks, certain kinds of bugs keep reappearing. The Top 10 highlights those most common and dangerous flaws. Addressing them is a crucial first step toward safer web coding practices [4].

Broken Access Control (A01:2021)

Broken Access Control is when an application fails to enforce who can see or do what. In practice, attackers can end up accessing accounts, admin panels, databases or other sensitive data without permission [5]. It's like a building where the door locks don't work: anyone can wander into someone else's room or a manager's office if the system doesn't check permissions properly.

> **Real-world example:** Imagine a website where your profile data is reached via a link like `example.com/user?id=1234`. If you change the number to someone else's ID (for example, `1235`), and the site doesn't double-check your rights, you suddenly see another person's data. That's Broken Access Control in action.

- **Least privilege:** Give users only the permissions they actually need (no more).
- **Server-side checks:** Enforce permissions on the backend for each request (don't rely only on hidden links or UI controls).
- **Role-based access:** Assign clear roles (user, admin, etc.) and make sure the code enforces them.
- **Secure defaults:** Turn off unused pages or services, remove default passwords, and avoid directory listings.

> **Infographic idea:** An image of a hallway with locked vs. unlocked doors could illustrate how proper access checks keep people out of restricted rooms.

Cryptographic Failures (A02:2021)

Cryptographic Failures occur when sensitive data (like passwords, credit card numbers, or personal details) is stored or sent without proper encryption [6]. In everyday terms, it's as if you wrote your credit card number and PIN on a postcard instead of in a sealed envelope. Anyone intercepting that postcard could read your data. Similarly, if a website uses weak or outdated encryption, attackers who eavesdrop or hack the database can easily steal confidential information.

Real-world example: A site sends login cookies over an unsecured connection, or stores passwords using a weak hash. An attacker snooping on the network or breaching the database can then intercept or crack those secrets.

- **Encrypt data in transit:** Always use HTTPS (TLS) so that data traveling between users and the server is encrypted.
- **Encrypt data at rest:** Store sensitive information (passwords, personal data) using strong, modern algorithms (e.g. AES for data, bcrypt or Argon2 for passwords).
- **Use proven libraries:** Don't write your own crypto – use well-reviewed libraries and keep them updated.
- **Key management:** Keep encryption keys safe (don't hardcode them or share them). Rotate or refresh keys regularly.

Infographic idea: Show two scenarios—one with plain text data flowing over a network, and one with encrypted data—highlighting how encryption protects information.

Injection (A03:2021)

Injection flaws happen when an application sends user input to an interpreter (like a database, command shell, or LDAP server) and that input includes malicious commands [7]. In other words, the attacker "injects" code into the system by crafting special input. It's like someone slipping secret instructions into your coffee order. If the barista (the app) thinks it's all just part of the order, they might do something unexpected.

Real-world example: A web form that builds a SQL query by concatenating strings. If an attacker enters something like `\' OR \'1\'=\'1`, the application might turn a login query into:

```
SELECT * FROM users WHERE username='' OR '1'='1';
```

Because `'1'='1'` is always true, the attacker could bypass login and gain access.

- **Validate and sanitize inputs:** Check and clean all user input on the server. Reject or escape special characters before using the data.
- **Use parameterized queries:** Do not directly concatenate user data into commands. Use prepared statements or ORM frameworks that separate data from code.
- **Use safe APIs:** Whenever possible, use high-level functions that don't invoke interpreters with raw user data.
- **Limit database permissions:** Even if injection happens, giving the app only read-only permissions can reduce damage.

Infographic idea: A graphic comparing an unsafe raw query (with user input embedded) versus a safe parameterized query, showing how the malicious part is neutralized.

Insecure Design (A04:2021)

Insecure Design means failing to consider security during the planning phase [8]. In other words, the app was never designed with proper defenses. Think of it like building a car with no brakes or seat belts – no amount of later patching can fix that fundamental flaw. These are not coding bugs per se, but missing security features or protections from the very beginning.

Real-world example: A developer builds a social networking feature (like tagging friends) without thinking about abuse. If there's no check on how data flows, attackers might exploit it to spread malicious links through friend invitations. Because the design never considered abuse cases, fixing it is hard after the fact.

- **Secure by design:** Start every project by discussing security. Use threat modeling (consider what could go wrong) and secure design patterns.
- **Security requirements:** Include security checks in your requirements and user stories (for example, "only team members can create new users").
- **Use tested frameworks:** Leverage well-designed libraries or patterns for authentication, authorization, and data handling instead of inventing your own.
- **Review designs:** Have security experts review design documents or user flows to catch weaknesses before coding.

Infographic idea: Show a comparison between a blueprint of a secure vs. insecure system (e.g. a well-planned building vs. one with missing supports), highlighting how missing security planning leads to problems.

Security Misconfiguration (A05:2021)

Security Misconfiguration happens when default or sloppy settings leave a system open to attack [9]. This can be something like leaving admin pages exposed, using factory-default passwords, or failing to apply security patches. It's like leaving your house key under the welcome mat – a hacker can easily walk in if you haven't locked the doors properly.

Real-world example: A development server is left in "debug" mode with verbose error messages visible, or the admin login page is using the default password. These simple oversights are very common and can let attackers seize control of your site.

- **Keep software up to date:** Regularly apply updates and patches to your server, OS, and libraries.
- **Remove defaults:** Change default passwords/API keys, and delete or disable any test code, sample apps, or unused services.
- **Harden configuration:** Use secure configuration guides (close unused ports, disable directory listings, set file permissions properly).
- **Separate environments:** Keep development, testing, and production environments separate (so mistakes in one don't affect the others).
- **Automate scans:** Use automated tools to scan for misconfigurations (for example, scanning for open ports or default credentials).

Infographic idea: A picture of a castle or fortress with an unlocked gate can highlight what happens when security settings are left open.

Vulnerable and Outdated Components (A06:2021)

Using outdated or unpatched components (software libraries, frameworks, etc.) can introduce serious risks [10]. It's like building a car and using an old tire that you know has defects—attackers love to target known weaknesses. Many popular libraries (for example, older versions of a web framework or plugin) may have public vulnerabilities that hackers exploit if they aren't updated.

> **Real-world example:** A website uses an older version of a content management system or JavaScript library that has a publicly known vulnerability. Attackers can scan for sites running that old version and break in with one click using tools that exploit that specific weakness.

- **Inventory and scan:** Keep a list of all software components and libraries your app uses, and check them regularly with vulnerability scanning tools.
- **Update regularly:** Apply patches and updates for your dependencies as soon as they are available (especially security patches).
- **Remove unused code:** Delete libraries or plugins that you don't use anymore. Fewer components mean fewer potential holes.
- **Use trusted sources:** Download packages from official repositories, and consider using signed versions or checksums to verify integrity.
- **Monitor advisories:** Stay informed about vulnerabilities in the tools you use (for example, subscribe to security bulletins for your framework or language).

> **Infographic idea:** A graphic showing a piece labeled "Software" with arrows pointing to out-of-date version numbers, highlighting how old components become targets.

Identification and Authentication Failures (A07:2021)

Authentication failures mean weaknesses in how users log in or prove who they are [11]. If attackers can bypass login screens or steal credentials, they can pretend to be legitimate users. It's like a club bouncer who fails to check IDs: anyone with a convincing story (or fake ID) can get in. Common issues include weak passwords, no multi-factor authentication (MFA), or poor session management.

> **Real-world example:** A site allows the same default password for all admin accounts, or it never logs users out after inactivity. Attackers can easily guess or steal credentials and then abuse the account. Even worse, if session tokens aren't protected, an attacker might hijack a logged-in user's session.

- **Strong passwords and policies:** Require long, unique passwords; avoid reuse. Consider password strength meters or guidelines.
- **Multi-factor authentication (MFA):** Whenever possible, add a second factor (like SMS or an authenticator app) especially for sensitive roles or admin access.
- **Secure session management:** Use secure cookies, set short session timeouts, and regenerate session IDs on login. Invalidate sessions on logout.
- **Monitor logins:** Detect and throttle repeated failed login attempts (account lockout or CAPTCHAs).
- **No default credentials:** Do not use default usernames/passwords (e.g. "admin/admin"). Change them on first use.

Infographic idea: Illustrate a login screen with multiple checks (password, phone code) versus one with only a weak password, showing how MFA adds an extra lock.

Software and Data Integrity Failures (A08:2021)

Software and Data Integrity Failures refer to trusting data or code without verifying its integrity [12]. In plain terms, it's like receiving a package from a friend but never checking if it was tampered with. This can happen, for example, if an application automatically installs updates or code modules without confirming they're legitimate. Attackers can insert malicious code into updates or dependencies (a supply-chain attack), and the system blindly trusts it.

Real-world example: A software project pulls a library from an external repository. If that library was maliciously modified (and the project isn't verifying checksums or signatures), the harmful code becomes part of the app. Similarly, installing updates from an unverified source could let an attacker take control.

- **Check digital signatures:** Use signed packages and verify signatures before installing updates or dependencies.
- **Trusted sources only:** Pull software only from official, trusted repositories (and pin versions so you don't automatically jump to unknown releases).
- **Secure CI/CD:** Protect your build and deployment pipeline (use access controls, don't expose API keys in public scripts).
- **Integrity checks:** Use checksums or hashes to verify critical data hasn't changed unexpectedly.
- **Minimal privileges:** Run automatic update processes with limited rights, so even if compromised they do less harm.

Infographic idea: A visual of an unlocked delivery box vs. a sealed, signed box could illustrate the need to verify what's inside software updates.

Security Logging and Monitoring Failures (A09:2021)

Logging and Monitoring failures mean not keeping track of important events or not noticing attacks when they happen. It's like having a security camera system but never checking the footage. If there are no logs or no one watches them, a breach can go on for a long time without detection. In fact, attackers often go undetected for months because their activity isn't spotted [13].

Real-world example: Imagine an online store that logs errors and login attempts, but the administrators never review them. Attackers could try repeated logins or exploit something and the site never raises an alarm. By the time the breach is noticed (if ever), a lot of damage could be done.

- **Enable detailed logs:** Record important events like failed logins, access control denials, data changes, and errors.
- **Centralize and protect logs:** Send logs to a secure, separate location or service so attackers cannot easily erase them.
- **Automate alerts:** Use monitoring tools to alert on suspicious patterns (e.g., many failed logins, unusual data downloads, or changes to admin accounts).

- **Regular reviews:** Periodically review your logs or use dashboards to look for signs of trouble.
- **Plan for incidents:** Have an incident response plan so your team knows what to do if something is detected.

> **Infographic idea:** A flowchart of an attack happening vs. no alerts (a silent breach), contrasted with an alert popping up on a monitoring screen for the same activity.

Server-Side Request Forgery (SSRF) (A10:2021)

Server-Side Request Forgery (SSRF) vulnerabilities occur when an application fetches a resource (like a URL or web page) on behalf of a user, and that input isn't properly validated [14]. An attacker can exploit this by tricking the server into requesting internal resources or services the attacker shouldn't have access to. It's as if someone at a help desk makes a request for you, but you secretly tell them to go into a secure area.

> **Real-world example:** A web app lets users enter a URL to fetch an image. The attacker inputs an internal URL like `http://localhost/admin` or a cloud metadata address (e.g. AWS `169.254.169.254`). Because the server is allowed to fetch those by default, it ends up returning sensitive information (like internal config or credentials) to the attacker.

- **Whitelist or validate hosts:** Only allow requests to known safe domains or IP ranges (never let user input fully control the target).
- **Block internal IPs:** Disallow requests to internal or reserved addresses (e.g. 127.0.0.1 or private ranges) in your code or network firewall.
- **Limit response data:** If you must fetch user-provided URLs, strip out any sensitive data from the response before showing it.
- **Timeouts and size limits:** Set timeouts and size limits on server requests to avoid denial-of-service via large or slow responses.
- **Keep libraries updated:** Use the latest HTTP request libraries or frameworks that include SSRF protections.

> **Infographic idea:** Show a trusted server acting as a proxy: illustrate an attacker "writing a note" to the server ("go fetch X"), and the server unwittingly accessing an internal, secure site.

Conclusion

In summary, the OWASP Top 10 is a friendly roadmap to the most common web security risks. We covered how each vulnerability works in simple terms – with examples like unlocked doors (Broken Access Control), unsealed letters (Cryptography), or an unwitting server used as a proxy (SSRF). By understanding these issues and following basic precautions (strong passwords, updates, input validation, etc.), even developers and users with limited technical experience can help make web applications much safer.

Call to action: OWASP's tools and documentation are free and open to everyone [15]. Check out the official OWASP Top 10 guide or try basic security checklists, and consider joining a local OWASP chapter. By staying aware and sharing knowledge, we all contribute to a more secure Internet for everyone [15].

Sources: OWASP Top 10 documentation and related educational sites [1] [2] [16] [5] [7] [8] [9] [10] [11] [17] [13] [14] [15].

[1] [2] [15] What Is OWASP? | Open Worldwide Application Security Project | Akamai
https://www.akamai.com/glossary/what-is-owasp

[3] [4] [16] OWASP Top Ten | OWASP Foundation
https://owasp.org/www-project-top-ten/

[5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [17] OWASP Top 10 Vulnerabilities in 2021 | Indusface Blog
https://www.indusface.com/blog/owasp-top-10-vulnerabilities-in-2021-how-to-mitigate-them/