What is OWASP Top 10?

OWASP Top 10 is a publicly shared standard awareness document for developers of the ten most critical web application security vulnerabilities, according to the Foundation. OWASP understands that a security vulnerability is any weakness that enables a malevolent actor to cause harm and losses to an application's stakeholders (owners, users, etc.).

The OWASP Top 10 list is developed by web application security experts worldwide and is updated every couple of years. It aims to educate companies and developers on how to minimize application security risks.

The latest update of the list was published in 2021, whereas the previous update was in 2017.

The OWASP list is also under development for mobile applications.

Next to the Top 10 list, OWASP also publishes and maintains the following resources:

- OWASP Testing Guide: "Best Practices" for application testing
- OWASP Juice Shop: An intentionally insecure web application for security training

The OWASP Top 10 2021 Web App Security Risks

Broken Access Control A01:2021

Access control refers to the enforcement of restrictions on authenticated users to perform actions outside of their level of permission. Broken access control occurs when such restrictions are not correctly enforced. This can lead to unauthorized access to sensitive information, as well as its modification or destruction.

Some of the common vulnerabilities of access control are:

- Granting free access to roles, functions, and capabilities where it should be limited by the principle of least privilege and denied by default
- Tampering with parameters and force browsing (i.e. modifying a URL or the HTML page) or modifying API requests to avoid access control checks
- Providing insecure direct object references (i.e. a unique identifier) that allows a user's account to be viewed and modified
- Elevation of privilege due to a bug or design flaw
- Tampering with metadata, such as JWT access control tokens, cookies, hidden fields, and abusing JWT invalidation
- Force browsing to access authenticated or privileged pages
- Allowing API access from untrusted/unauthorized sources due to a Cross-Origin Resource Sharing (CORS) misconfiguration
- Accessing API with no POST, PUT, and DELETE access controls in place

How to prevent broken access control?

Access control must be implemented in code on a trusted server or a serverless API to be effective. This reduces the chance of an attacker tampering with the access control check or metadata. In addition, OWASP recommends the following measures to handle broken access control:

- With the exception of public resources – deny by default
- Enforce record ownership, rather than granting any user access to any record
- Introduce access control mechanisms once and re-use them repeatedly in the application
- Domain models should enforce unique application business limit requirements
- Monitor and record failed access control attempts and alert administrators when necessary
- Invalidate stateful session identifiers on the server after logout
- Makes stateless JSON web tokens (JWT) short-lived
- Ensure that no metadata (git) nor backup files are present in web roots
- Disable web server directory listing
- Reduce the effect of automated attack tools by rate-limiting API and controller access
- Perform functional access control unit and integration tests

On top of these measures, you may also want to:

- Delete any inactive or unnecessary accounts
- Use multi-factor authentication at all access points
- Shut down unnecessary access points
- Enforce the principle of least privilege (PoLP)
- Eliminate services that are not needed on your server

Cryptographic Failures A02:2021

Cryptographic failures refer to problems with cryptography or the absence of cryptography altogether. Previously this item was known as Sensitive Data Exposure, but this name was not entirely accurate as it described a symptom and effect rather than a cause. Cryptographic failure may and often does lead to exposure of data.

This type of failure applies to the protection and secrecy of data in transit and at rest. Such data typically include authentication details, such as usernames and passwords, but also personally identifiable information (PII) such as personal and financial information, health records, business secrets, and more.

Failures arise for a variety of reasons, and vulnerabilities are often exploited via a man-in-the-middle attack. Common reasons for cryptographic shortcomings include:

- Storing or transmitting sensitive data in clear text
- Using outdated or weak cryptographic algorithms and protocols
- Using default or weak crypto keys, not using key management and rotation
- Not enforcing encryption
- Not properly validating the server certificate and the trust chain
- Ignoring or reusing initialization vectors or using an insecure mode of operation

These are some of the vulnerabilities that attackers can exploit to gain access to sensitive data.

How to prevent cryptographic failures?

According to OWASP, the proactive measures that companies and organizations can take to prevent cryptographic failures include:

- Classify data (processed, stored, or transmitted) that is transmitted by the application and identify which data is sensitive according to privacy laws, regulations, and business needs
- Implement security controls depending on data classification
- Don't store sensitive data, discard it as soon as possible, or use PCI DSS compliant tokenization or even truncation
- Encrypt all data at rest that needs to be stored
- Use strong algorithms, protocols, and keys
- Use the Transport Layer Security (TLS) protocol with forward secrecy to encrypt all data in transit
- Use HTTP Strict Transport Security (HSTS) directive encryption or similar
- Never use FTP and SMTP protocols to transfer sensitive data
- Disable caching for user responses that contain sensitive data
- Use functions that always salt and hash passwords and have a work factor such as brcrypt, scrypt, Argon2, PBKDF2 to store passwords
- Carefully choose the initialization vectors, depending on the mode of operation – for many this may mean a cryptographically secure pseudo-random number generator (CSPRNG).
- Always use authenticated encryption
- Use cryptographic random key generation and store the keys as byte arrays
- Use cryptographic randomness where needed but ensure it is not seeded predictably or with low entropy

Injection A03:2021

An injection attack refers to untrusted data by an application that forces it to execute commands. Such data or malicious code is inserted by an attacker and can compromise data or the whole application. The most common injection attacks are SQL injections and cross-site scripting (XSS) attacks, but code injections, command injections, CCS injections, and others.

An application is vulnerable to an injection attack when one or several of the following conditions are present:

- Data supplied by users is not validated, filtered, or sanitized
- The interpreter directly uses dynamic queries or non-parameterized calls without context-aware escaping
- Hostile data is used directly, concatenated, or used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.

When an injection attack is successful, the attacker can view, modify or even delete data and possibly gain control over the server.

How to prevent an injection?

The main vulnerability to an injection attack is data not being kept separate from commands and queries. Here is what can be done to prevent an injection:

- The most common solution is the use of a safe API that avoids the use of the interpreter altogether or uses parameterized queries or migrates to Object Relational Mapping Tools (ORMs). Note that parametrized stored procedures can still be vulnerable to an SQL injection if queries and data are concatenated or hostile data is executed with EXECUTE IMMEDIATE or exec()
- Use positive server-side input validation (whitelist) – this is a partial defense since many applications require the use of special characters
- For residual dynamic queries, special characters can be avoided via the specific escape syntax for that interpreter
- Use database controls within queries such as LIMIT to prevent mass exposure of data if an SQL injection is successful
- Run a SQL Injection Test or XSS Test by an atomated vulnerability tool such as Crashtest Security Suite (now 14-day Free Trial in it's full version), which will inform you about every weaknesses and attacks vectors discovered in your web app or API.

Insecure Design A04:2021

This category of vulnerabilities is focused on the risks associated with flaws in design and architecture. As explained by OWASP, these are different from the risks associated with deficiencies in implementation. A well-implemented insecure design is still vulnerable to attacks.

Insecure design refers, in part, to the lack of security controls and business risk profiling in the development of software, and thereby the lack of proper determination of the degree of security design that is needed.

How to prevent insecure design?

- Implement a secure development lifecycle with application security experts to assess the design security and privacy-related requirements
- Introduce and use a secure design pattern library or paved road ready to use components
- Apply threat modeling methods to critical authentication, access control, business logic, and key flows
- Make security language and controls a part of user stories
- Enforce plausibility checks from frontend to backend at each tier of the system
- Validate critical flows' resistance to the threat model via unit and integration tests. Compile use and misuse cases for every tier of the application
- Separate tier layers and network layers on the system, based on the exposure and protection requirements
- Separate tenants robustly by design throughout all tiers
- Restrict resource consumption by user or service

Security Misconfiguration A05:2021

Security misconfiguration refers to security controls that are not secured or not configured correctly. This vulnerability is frequently due to one of the following reasons:

- Lacking security hardening in any part of an application

- Wrongly configured permissions on cloud services
- Unnecessary features, such as ports, services, pages, accounts, or privileges are allowed or installed
- Default accounts/passwords are enabled or unchanged
- Error messages displayed to users contain stack traces or other sensitive information
- The latest security features are not enabled or implemented correctly
- The server, framework, libraries, or databases' security settings are not set to secure values
- Security headers or directives are not sent by the server or are not set to secure values
- The software is not up to date

An automated pentest tool such as Crash Test Security can detect application vulnerabilities that may open the door to an attack due to security misconfigurations. Sign up for a free trial and start your first vulnerability scan in minutes.

How to prevent security misconfiguration attacks?

OWASP recommends the implementation of secure installation processes. Along with that, you should:

- Securing the system via a hardening process by developing and automating the process of deploying a separate and secure environment. Each separate environment (development, QA, production) must be configured identically but be accessible via different credentials
- Use a "minimal platform" that has no unnecessary features, components, documentation, or samples. Do not install or remove unnecessary features and frameworks
- Review the cloud storage permissions
- Introduce a review and update of the configurations of all security notes, updates, and patches as part of the patch management process
- Implement segmented application architecture via segmentation, containerization, or cloud security groups to separate tenants and components
- Security directives, such as security headers, should be sent to clients
- Automate the verification of the effectiveness of configurations and settings in each environment

Vulnerable and Outdated Components A06:2021

This category was previously known as “Using Components with Known Vulnerabilities.” Component vulnerabilities can arise in one of the following situations:

- If you are not aware of the versions of client-side and server-side components that you use
- If the software is vulnerable, unsupported, or out of date. This includes the operating systems, web/application server, database management system (DBMS), applications, APIs and any components, runtime environments, and libraries
- If you are not scanning for vulnerabilities regularly and following security news about the components that you use
- If you do not fix or upgrade your platform, framework, and dependencies when patches come out
- If your developers are not performing tests on the compatibility of updated, upgraded, or patched libraries
- If your components’ configurations are not secured

How to avoid the use of vulnerable or outdated components?

To avoid the dangers associated with the use of such components, you should:

- Remove any unused dependencies, unnecessary features, components, files, and documentation
- Regularly inventory the versions of client-side and server-side components, as well as their dependencies through the use of OWASP Dependency-Check, retire.js, etc. Follow sources like Common Vulnerability and Exposures (CVE) and National Vulnerability Database (NVD), and subscribe to email alerts to receive news about any vulnerabilities in components. To automate the process, utilize software composition analysis tools.
- Use only official sources and secure links to obtain components. Opt for signed packages.
- Keep an eye out for libraries and components that are not being maintained and do not have security patches for old versions. If you cannot patch, deploy a virtual patch to monitor, detect, or protect against the known vulnerability

Identification and Authentication Failures A07:2021

This set of vulnerabilities was previously known as "Broken Authentication." These may arise if an application:

- Is not protected against automated attacks such as credential stuffing
- Allows for brute force attacks
- Accepts the use of default, weak, or well-known passwords
- Has weak or ineffective credential recovery and forgotten password procedures
- Employs plain text, encrypted, or weakly hashed password data stores
- Does not use or has ineffective multi-factor authentication
- Exposes the session identified in the URL
- Reuses the session identified after login
- Does not properly invalidate user sessions and authentication tokens during logout or when inactive

How to avoid identification and authentication vulnerabilities?

The prevention measures suggested by OWASP include:

- Implement multi-factor authentication, whenever possible, to hinder credential stuffing, brute force, and stolen credential reuse attacks
- Never ship or deploy with default credentials, in particular for admin-level users
- Perform weak password checks
- Use the National Institute of Standards and Technology (NIST) 800-63b guidelines or other current and evidence-based password policies to determine password length, complexity, and rotation policies
- Harden registration, credential recovery, and API pathways against account enumeration attacks through the use of identical messages for all outcomes
- Limit or progressively delay repeated login attempts after failure but avoid creating a denial of service scenario. Log and monitor failed attempts and notify admins if credential stuffing, brute force, or another type of attack is detected
- Utilize a server-side, secure, and built-in session manager that generates new random session IDs with high entropy after login. Do not keep the session identified in the URL, and make sure it is securely stored and invalidated after logout, idle, and absolute timeouts

Software and Data Integrity Failures A08:2021

This new category on the OWASP list relates to vulnerabilities in software updates, critical data, and CI/CD pipelines whose integrity is not verified.

For example, an application that relies on plugins, libraries, or modules from unverified and untrusted sources, repositories, or content delivery networks (CDNs) may be exposed to such a type of failure.

A similar source of failure may be the auto-update functionality of most applications that do not necessarily include a thorough integrity check. This leaves the door open for attackers to distribute their updates that are intended to create vulnerabilities.

Finally, this category also includes what was previously called “Insecure Deserialization” in the 2017 list. Failures that arise here are due to objects or data encoded or serialized into a structure that is visible to an attacker and which they can modify.

How to prevent software and data integrity failures?

To protect against insecure deserialization, OWASP recommends the following steps:

- Use mechanisms such as digital signatures to verify that software or data from a source has not been tampered with
- Make sure libraries and dependencies are only using trusted repositories or consider hosting an internal known-good repository that’s vetter if you have a higher risk profile
- Use a software supply chain security tool to make sure that components do not have any known vulnerabilities
- Institute a review process for code and configuration changes to reduce the risk of malicious code or configurations being inserted into your software pipeline
- Implement thorough segregation, configuration, and access control to your CI/CD pipeline to guarantee the integrity of the code that flows through the build and deploy processes
- Make sure no unsigned or unencrypted serialized data is sent to untrusted clients without prior integrity check or digital signature to detect tampering or replay of the data

Security Logging and Monitoring Failures A09:2021

Previously known as "Insufficient Logging & Monitoring," this category has been expanded to include more types of failures. While logging and monitoring are challenging to test, this category is essential because failures can impact accountability, visibility, incident alerting, and forensics.

Logging, detection, monitoring, and operational response failures occur when:

- Logins, failed logins, high-value transactions, and other types of auditable events are not logged
- Inadequate, unclear, or no messages are generated by warnings and errors
- API and application logs are not examined for suspicious activity
- You only store logs locally
- Alerting thresholds and response escalation processes have not been instituted or are not effective
- Alerts are not triggered by penetration testing or scans by dynamic application security testing tools
- The application cannot detect, escalate, or alert for active attacks in real-time or near real-time

Your application can further be exposed to information leakage if logging and alerting events are visible to users or attackers.

How to prevent security logging and monitoring failures?

To prevent the vulnerabilities that can arise due to these failures, OWASP recommends that some of the following controls be implemented, depending on the severity of the risk:

- Log all login, access control, and server-side input validations failures with sufficient user context to spot suspicious or malicious accounts. Store them long enough to perform delayed forensic analysis

- Make sure logs are in a format that can be easily consumed by log management solutions
- Avoid injections or attacks on the logging or monitoring systems by encoding log data properly
- Implement an audit trail with integrity controls for high-value transactions such as append-only database tables to prevent tampering or deletion
- Establish effective monitoring and alerting to detect suspicious activities and respond to them fast
- Introduce or adopt an incident response and recovery plan, such as National Institute of Standards and Technology (NIST) 800-61r2 or later

Server-Side Request Forgery (SSRF) A10:2021

Server-side request forgery issues arise when a web application does not validate the user-supplied URL when fetching a remote resource. This enables attackers to force the application to send a crafted request to an unexpected destination, even if protected by a firewall, VPN, or some other type of network access control list (ACL).

Fetching a URL is a common feature among modern web applications, which results in increases in instances of SSRF. Moreover, these are also becoming more severe due to the increasing complexity of architectures and cloud services.

How to prevent server-side request forgery?

OWASP suggests several different courses of action for preventing SSRF. These include implementing defense-in-depth controls in one or several layers.

Prevention at the network layer

- Reduce the impact of SSRF by segmenting the remote resource access functionality in separate networks
- Block all but essential traffic by instituting “deny by default” network policies or network access control rules

    - Introduce ownership and a lifecycle for firewall rules depending on the application
    - Log all accepted and blocked network flows on firewalls

Prevention at the application layer:

- All client-supplied input data must be sanitized and validated
- Use a positive allow list to enforce the URL schema, port, and destination
- Do not send raw responses to clients
- Disable HTTP redirections
- Avoid attacks such as DNS rebinding and “time of check, time of use” (TOCTOU) race conditions by verifying the URL consistency
- Do not use a deny list or regular expression to mitigate SSRF – these can be bypassed in a variety of ways

Other possible measures:

- Control local traffic on front systems instead of deploying other security services
- For very high protection needs use network encryption on independent systems with frontends that have dedicated and manageable user groups