nonamesecurity.com

OWASP Top 10: Definition and Related FAQs | Noname Security

21-26 minutes

What are the OWASP Top 10 Risks?

Over the past few years, the OWASP 10 has been updated several times. The OWASP Top 10 list for 2021 is the most data-driven version yet. Data from over 500,000 applications form the basis for this update—making it the largest application security data set.

Web application security risks have evolved since 2017. OWASP Top 10 2021 combined several categories and created and added three: Software and Data Integrity Failures, Insecure Design, and Server-Side Request Forgery.

There were also changes in ordering on the list.

Broken Access Control leads the OWASP Top 10 list for 2021, listed at five in 2017. Fully 94 percent of tested applications had some form of Broken Access Control, more than any other category. Along these lines, Injection Flaws, formerly ranked first, has dropped to third, and Cross-Site Scripting, formerly listed at seven, has been included within Injection Flaws.

Finally, a few categories have been renamed for accuracy; for example, Broken Authentication has been renamed Identification and Authentication Failures, and now includes CWEs (Common Weakness Enumerations) that are more related to identification failures. This category has dropped from number two in 2017 to seventh place in 2021.

Broken Access Control

Access control refers to permission levels for authenticated users and enforcing related restrictions on actions outside those levels. When there is a failure to enforce those restrictions correctly, broken access control occurs, potentially allowing unauthorized access to sensitive information, and possibly causing its destruction, modification, or loss.

Common access control vulnerabilities:

- Granting unrestricted access to functions, roles, and capabilities that should be denied by default and limited by the principle of least privilege
- Allowing user accounts to be viewed or modified by providing unique identifiers or other insecure direct object references
- Avoiding access control checks by modifying API requests or tampering with parameters and force browsing

- Abusing JWT invalidation and tampering with metadata, such as cookies, JWT access control tokens, and hidden fields
- Design flaws and bugs that elevate privilege inappropriately
- Cross-Origin Resource Sharing (CORS) misconfigurations that allow API access from untrusted data and unauthorized sources
- Accessing privileged or authenticated pages via force browsing
- Accessing API without implementing POST, PUT, and DELETE access controls

  How to prevent broken access control? To be effective, implement access control in code on a serverless API or a trusted server. This reduces the opportunities for attackers to tamper with metadata or the access control check.

  OWASP recommends handling broken access control with the following measures:

- Deny by default, except for public resources

- Introduce and then reuse access control mechanisms repeatedly in the application
- Rather than granting any user access to any record, enforce record ownership
- Monitor for attempts to gain access and record failed access control attempts; alert administrators as needed
- Unique business limit requirements for applications should be enforced by domain models
- Ensure stateless JSON web tokens (JWT) are short-lived
- After logout, invalidate stateful session identifiers on the server
- Disable web server directory listing
- Ensure web roots contain neither backup files nor metadata (git)
- Conduct integration and functional access control unit tests
- Use controller access and rate-limiting API to reduce the effect of automated attack tools

Other measures to try may include:

- Delete unnecessary or any inactive accounts
- Shut down unnecessary access points
- Implement multi-factor authentication (MFA) at all necessary access points
- Eliminate unneeded services on the server
- Enforce the principle of least privilege (PoLP)

Cryptographic Failures

Cryptographic failure, previously classified as Sensitive Data Exposure, involves the absence of cryptography or problems with cryptography. Cryptographic failure can and sometimes does lead to sensitive data exposure, but this is not the root cause, but the effect of the cryptographic

issue.

This type of cryptographic failure involves the secrecy and protection of data, both at rest and in transit. Such data generally include normal authentication details, such as passwords and usernames, as well as personally identifiable information (PII) such as financial details, personal information, business secrets, health records, and more.

Cryptographic failures often happen when a man-in-the-middle attack exploits a vulnerability. Common cryptographic flaws include:

- Using cryptographic protocols and algorithms that are weak or outdated
- Transmitting or storing sensitive data in unencrypted form
- Failing to enforce encryption
- Failing to use key rotation and management, using weak or default crypto keys
- Using an insecure operation mode or reusing or

ignoring initialization vectors

- Failing to properly validate the trust chain and server certificate

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

  How to prevent cryptographic failures?

  According to OWASP, there are many proactive measures that companies and organizations can take to prevent cryptographic failures.

  These OWASP data protection measures include:

- Classify data the application sends as stored, processed, or transmitted and identify which data is sensitive based on regulations, privacy laws, and business needs
- Implement security controls based on data classification
- Encrypt all data at rest for storage
- Discard sensitive data as soon as possible rather

than storing it, or use PCI DSS compliant tokenization or truncation to keep it safer

- Encrypt all data in transit using the Transport Layer Security (TLS) protocol with forward secrecy
- Use strong protocols, algorithms, and keys
- Never transfer sensitive data with SMTP or FTP protocols
- Use HTTP Strict Transport Security (HSTS) directive encryption
- Always salt and hash passwords for storage using functions that have a work factor such as scrypt, brcrypt, PBKDF2, or Argon2
- User responses that contain sensitive data should have caching disabled
- Always use authenticated encryption
- Select initialization vectors carefully based on operational mode such as a cryptographically secure pseudo-random number generator (CSPRNG).
- Deploy cryptographic randomness where

essential, but do not seed it with low entropy or predictably

- Use cryptographic random key generation
- Store cryptographic keys as byte arrays

Injection

During an injection attack, an attacker inserts malicious code or data into an application that forces the app to execute commands. Such code can compromise core data or the application as a whole. Cross-site scripting (XSS) attacks and SQL injections are the most common injection attacks, but there are others, including command injections, code injections, and CCS injections.

Several conditions can render an application vulnerable to an injection attack:

- User data is not filtered, validated, or sanitized
- The interpreter uses dynamic queries or non-parameterized calls directly without context-aware escaping
- The system extracts additional, sensitive records by using hostile data directly

  A successful injection attack allows an attacker to modify, view, or even delete data and potentially gain control of the server.

  How to prevent an injection? Failing to keep data separate from queries and commands is the main vulnerability to an injection attack.

  Prevent an injection as follows:

- Use a safe API that avoids the interpreter entirely, migrates to Object Relational Mapping Tools (ORMs), or uses parameterized queries. If data or queries are concatenated or an EXECUTE IMMEDIATE or exec() command are used to execute hostile data, parameterized stored procedures may remain vulnerable to an SQL injection

- Special characters can be avoided for residual dynamic queries via the specific escape syntax for that interpreter
- Use a whitelist or positive server-side input validation as a partial defense since special characters are required for many applications
- In the event of a successful SQL injection, prevent mass exposure of data by using database controls within queries such as LIMIT

Insecure Design

This new category emphasizes securing applications by integrating OWASP API security into software design early in the application development cycle to avoid risks from architecture and design flaws. Insecure design references a lack of business risk profiling and

security controls in software development, which results in improper determination of the optimal degree of security design. Deficiencies in implementation are different from design insecurity, because an insecure design, even one that is well-implemented, remains vulnerable to attacks.

Common mitigation techniques for insecure design rely on baking application security into software development from the outset and on shift-left security. Development and security teams should start considering means of exploitation and how potential threat actors will attack as early as possible, and integrate OWASP threat modeling into development and planning processes to better prepare for all possible outcomes.

How does OWASP recommend developers prevent insecure design?

- Assess privacy-related requirements and design security as part of a secure development lifecycle and implement that lifecycle with application

security experts part of the team (see the OWASP SAMM or software assurance maturity model for more information)

- Use ready to use components or a secure design pattern library
- Apply OWASP threat modeling techniques to critical access control, authentication, business logic, and key flows
- Ensure that security controls and security language are part of user stories
- Enforce frontend to backend plausibility checks at each system tier
- Use integration and unit tests to validate resistance of critical flows to the threat model; at each application tier, compile use and misuse cases
- Use protection and exposure requirements to separate network and tier layers on the system
- Use robust design to separate tenants throughout all tiers
- Restrict resource consumption by service or user

Security Misconfiguration

OWASP security misconfiguration refers to misconfigured or otherwise insecure security controls. The cause of this vulnerability is often one of the following issues:

- A lack of security hardening
- Cloud service permissions wrongly configured
- Allowing or installing unnecessary features such as accounts, pages, ports, privileges, or services
- Enabling default passwords or accounts or otherwise using them unchanged
- Including sensitive data such as stack traces in error messages displayed to users
- Updated but wrongly implemented or disabled security features

- Insecure values in the security settings for the framework, server, libraries, or databases
- Unsent or security headers or directives or settings on insecure values
- Out of date software

OWASP recommends businesses prevent security misconfiguration attacks by implementing secure installation processes and engaging in other best practices:

- Developing and automating the process of deploying each separate environment securely via a hardening process and configuring each to be accessible via different credentials yet identical otherwise
- Use a minimal platform with no unnecessary features, frameworks, documentation, components, or samples
- Review cloud storage permissions
- As part of patch management, review and update configurations of all patches, security notes, and updates

- Separate tenants and components via containerization, segmentation, or cloud security groups to implement segmented application architecture
- Send security headers and other directives to clients
- Automate the setting and configuration effectiveness verification in each environment

Vulnerable and Outdated Components

Component vulnerabilities, previously called "Using Components with Known Vulnerabilities," can arise in several cases:

- Different versions of server-side and client-side components in use

- Unsupported, vulnerable, or out of date software including web/application server, operating systems, applications, database management system (DBMS), runtime environments, APIs and any components, and libraries
- Failure to follow security news and updates and scan for OWASP vulnerabilities routinely
- Failure to upgrade or fix framework, platform, and dependencies as patches are released
- Developers do not perform compatibility tests on upgraded, updated, or patched libraries
- Unsecured configurations for components

OWASP recommends users avoid risks associated with the use of vulnerable or outdated components in several ways:

- Remove unnecessary files, components, features, and documentation and unused dependencies
- Use OWASP Dependency-Check and other OWASP tools to inventory versions of server-side and client-side components and their

dependencies regularly. Follow security and vulnerability news and check the OWASP tools list regularly

- Use software composition analysis tools to automate the process
- Opt for signed packages and use only secure links and official sources to obtain components
- Watch for unmaintained components and libraries and those that lack security patches for old versions. Deploy a virtual patch to detect, monitor, or guard against known OWASP vulnerabilities if you cannot patch

Identification and Authentication Failures

These vulnerabilities were formerly called “Broken Authentication.” OWASP broken

authentication and session management flaws can arise under several circumstances for an application:

- Brute force attacks are allowed by the application
- No protection from credential stuffing and other automated attacks
- Ineffective or weak forgotten password and credential recovery procedures
- Default, well-known, or weak passwords accepted by the app
- No or ineffective multi-factor authentication (MFA)
- Password data stores are plain text or weakly hashed
- Sessions identified after login are reused
- Sessions identified in URL are exposed
- Failure to properly invalidate authentication tokens and user sessions when inactive and during logout

Prevent authentication and identification vulnerabilities with these measures

suggested by OWASP:

- Prevent brute force, credential stuffing, and stolen credential reuse attacks as much as possible with multi-factor authentication
- Conduct checks for weak passwords
- Especially for admin-level users, never deploy with default credentials
- Use identical messages for all outcomes to harden credential recovery, registration, and API pathways against account enumeration attacks
- Generate new high entropy random session IDs after login using a secure, server-side, built-in session manager. Do not retain the identified session and after logout, idle, and absolute timeouts ensure it is securely invalidated and stored
- Set evidence-based password policies for password complexity, length, and rotation based on National Institute of Standards and Technology (NIST) 800-63b guidelines
- Limit repeated login attempts after failure or

progressively delay them while avoiding a denial of service problem. If the team detects brute force, credential stuffing, or another type of attack, notify admins and log and monitor failed attempts

Software and Data Integrity Failures

Another new 2021 category relates to security risks and vulnerabilities concerning unverified critical data, software updates, and CI/CD pipelines. For example, applications that rely on libraries, plugins, or modules from untrusted and unverified repositories, sources, or content delivery networks (CDNs) can experience this kind of failure.

Similarly, many applications have an auto-update functionality that does not include a thorough

integrity check. This paves the way for updates from attackers that create vulnerabilities. This category also covers failures that arise from what was formerly called “Insecure Deserialization.” In this case, attackers find data or objects that are visible and modify them, causing them to run unsafe dependencies from public repositories which results in failures.

Clearly, including integrity checks every time dependencies are downloaded is a good step to take. Downloading from only trusted sources by using private registries is an option for some users. It’s also possible to expose security issues by scanning dependencies as part of the CI/CD pipeline before the final deployment.

In addition, OWASP recommends the following steps to protect against insecure deserialization:

- Use digital signatures and similar mechanisms to verify that data or software from a source was not tampered with
- Check components for known OWASP

vulnerabilities with a software supply chain security tool

- Ensure dependencies and libraries use only trusted repositories or for those with a higher risk profile, host an internal known-good repository
- Guarantee code integrity through the build and deploy processes by implementing thorough configuration, segregation, and access control to your CI/CD pipeline
- Reduce the risk of malicious code or configuration insertions into the software pipeline by reviewing changes to code and configurations
- To detect tampering or replay of the data, send no unencrypted or unsigned, serialized, untrusted data to clients without first verifying the digital signature or conducting an integrity check

Security Logging and Monitoring Failures

In 2017, this category was called “Insufficient Logging and Monitoring,” and now it includes more kinds of failures such as detection and operational response failures.

These failures occur when:

- Auditable events such as high-value transactions, logins, failed logins, and others are not logged
- No examination of API and application logs for suspicious activity
- Errors and warnings generate unclear, inadequate, or no messages
- No or ineffective response escalation processes and/or alerting thresholds
- Logs stored locally only
- Application cannot monitor for or detect active OWASP attacks in real-time, or escalate and alert as needed
- Dynamic application security testing tool scans or penetration testing does not trigger alerts

OWASP recommends that users prevent these failures by implementing some of these controls, depending on risk level:

- Log and store all input validations failures affecting access control, logins, and the server side with enough user context to detect suspicious accounts or malicious activity and perform delayed forensic analysis on all such failures
- Encode log data properly to avoid attacks or injections on monitoring or logging systems
- Maintain logs in easily consumable format for management solutions
- Implement effective monitoring systems to detect suspicious activities with rapid alerting to respond promptly
- Prevent tampering or deletion in the context of high-value transactions by implementing an audit trail and use append-only database tables and other integrity controls
- Follow an incident response and recovery plan, for example National Institute of Standards and

Technology (NIST) 800-61r2

Server-Side Request Forgery (SSRF)

This new risk category focuses on server-side forgery attacks that force the server to issue forged HTTP requests on its behalf. These kinds of issues happen when a web application fetches remote resources without validating user-supplied URLs. Even if protected by a firewall, VPN, or some other type of network access control list (ACL), this failure allows attackers to force the application to send a forged request to an unexpected destination, increasing the chance of performing unintended actions or exposing

sensitive information.

It is common for modern web applications to fetch URLs, increasing the chances of SSRF. When requests trigger server hooks or events that perform any data manipulation or exfiltration, this type of attack tends to happen. Added complexity from cloud services and complex architectures are also making problems from these attacks more severe.

To prevent server-side request forgery attacks, always maintain a whitelist of domains with strict verification defined with outbound firewall rules or SSL pinning. Any deviations from these patterns should be disallowed.

OWASP also suggests implementing layered, defense-in-depth controls to prevent SSRF.

Prevention of SSRF at the network layer:

- Segment remote resource access functionality into distinct networks to reduce SSRF impact
- Institute network access control rules or “deny by default” network policies to block all but essential

traffic

- Depending on the app, introduce a lifecycle and ownership for firewall rules
- Log all blocked and accepted network flows on firewalls

Prevention of SSRF at the application layer:

- Sanitize and validate all client-supplied input data
- Disable HTTP redirections
- Enforce the port, URL schema, and destination with a positive allow list
- Do not send clients raw responses
- A regular expression or deny list are easily bypassed in multiple ways, so they cannot be used to mitigate SSRF
- Avoid “time of check, time of use” (TOCTOU) race conditions, DNS rebinding, and related attacks by verifying URL consistency

Other possible measures:

- Opt to control local traffic before deploying other security services
- Use with independent systems with dedicated and manageable user groups on the front ends protected by network encryption for very high protection needs

  These changes to the OWASP Top Ten reflect trends in application security and development. As demand for high-quality products continues to grow, developers introduce more cloud-native technologies to hasten application development cycles, and it becomes even more critical to bake scalable security into the plan from the outset.

  Andrew van der Stock, Executive Director at OWASP, discusses the new OWASP Top Ten 2021, the methodology behind it, the categories, the data collection and analysis process and how to start an AppSec Program with the OWASP Top 10.