VULSCAN DIGITAL SECURITY

OWASP API Security Top 10

A foundational element of innovation in today's app-driven world is the API. From banks, retail and transportation to IoT, autonomous vehicles and smart cities, APIs are a critical part of modern mobile, SaaS and web applications and can be found in customer-facing, partner-facing and internal applications. By nature, APIs expose application logic and sensitive data such as Personally Identifiable Information (PII) and because of this have increasingly become a target for attackers. Without secure APIs, rapid innovation would be impossible.

The OWASP API Security Top 10 focuses on strategies and solutions to understand and mitigate the unique vulnerabilities and security risks of Application Programming Interfaces (APIs).

API Security Challenges

APIs play a very important role in modern applications' architecture. Since creating security awareness and innovation have different paces, it's important to focus on common API security weaknesses.

The primary goal of the OWASP API Security Top 10 is to educate those involved in API development and maintenance, for example, developers, designers, architects, managers, or organizations.

It is important to realize that over the last few years, architecture of applications has significantly changed.

Currently, APIs play a very important role in this new architecture of microservices, Single Page Applications (SPAs), mobile apps, IoT, etc

How to use this cheat sheet

We have taken the time to summarize each individual vulnerability identified in the API Security Top 10 list. In each section we provide a use case for each vulnerability type as well as information on how to prevent it. When in doubt, please consider visiting the OWASP API Security project for more detailed guidance at https://owasp.org/www-project-api-security/.

A1: BROKEN OBJECT LEVEL AUTHORIZATION

Attacker substitutes ID of their resource in API call with an ID of a resource belonging to another user. Lack of proper authorization checks allows access. This attack is also known as IDOR (Insecure Direct Object Reference).

USE CASES

- API call parameters use IDs of resourced accessed by the API: `/api/shop1/financial_details`
- Attackers replace the IDs of their resources with different ones, which they guessed: `/api/shop2/financial_details`
- The API does not check permissions and lets the call through
- Problem is aggravated if IDs can be enumerated: `/api/123/financial_details`

HOW TO PREVENT

- Implement authorization checks with user policies and hierarchy.
- Do not rely on IDs that the client sends. Use IDs stored in the session object instead.
- Check authorization for each client request to access database.
- Use random IDs that cannot be guessed (UUIDs).

A2: BROKEN AUTHENTICATION

Poorly implemented API authentication allows attackers to assume other users' identities.

USE CASES

- Unprotected APIs that are considered "internal"
- Weak authentication that does not follow industry best practices
- Weak API keys that are not rotated
- Passwords that are weak, plain text, encrypted, poorly hashed, shared, or default passwords
- Authentication susceptible to brute force attacks and credential stuffing
- Credentials and keys included in URLs
- Lack of access token validation (including JWT validation)
- Unsigned or weakly signed non-expiring JWTs

HOW TO PREVENT

- Check all possible ways to authenticate to all APIs.
- APIs for password reset and one-time links also allow users to authenticate, and should be protected just as rigorously.
- Use standard authentication, token generation, password storage, and multi-factor authentication (MFA).
- Use short-lived access tokens.
- Authenticate your apps (so you know who is talking to you).
- Use stricter rate-limiting for authentication and implement lockout policies and weak password checks.

A3: EXCESSIVE DATA EXPOSURE

The API may expose a lot more data than what the client legitimately needs, relying on the client to do the filtering. If attackers go directly to the API, they have it all.

USE CASES

- The API returns full data objects as they are stored in the backend database.
- The client application filters the responses and only shows the data that the users really need to see.
- Attackers call the API directly and get also the sensitive data that the UI would filter out.

HOW TO PREVENT

- Never rely on the client to filter data!
- Review all API responses and adapt them to match what the API consumers really need.
- Carefully define schemas for all the API responses.
- Do not forget about error responses, define proper schemas as well.
- Identify all the sensitive data or Personally Identifiable Information (PII), and justify its use.
- Enforce response checks to prevent accidental leaks of data or exceptions.

A4: LACK OF RESOURCES & RATE LIMITED

The API is not protected against an excessive number of calls or payload sizes. Attackers can use this for Denial of Service (DoS) and authentication flaws like brute force attacks.

USE CASES

- Attackers overload the API by sending more requests than it can handle.
- Attackers send requests at a rate exceeding the API's processing speed, clogging it up.
- The size of the requests or some fields in them exceed what the API can process.
- "Zip bombs" - archive files that have been designed so that unpacking them takes excessive amounts of resources and overloads the API

.

HOW TO PREVENT

- Define proper rate limiting.
- Limit payload sizes.
- Tailor the rate limiting to be match what API methods, clients, or addresses need or should be allowed to get.
- Add checks on compression ratios.
- Define limits for container resources.

A5: BROKEN FUNCTION LEVEL AUTHORIZATION

The API relies on the client to use user level or admin level APIs as appropriate. Attackers figure out the "hidden" admin API methods and invoke them directly.

USE CASES

- Some administrative functions are exposed as APIs.
- Non-privileged users can access these functions without authorization if they know how.
- Can be a matter of knowing the URL, or using a different verb or a parameter:
  - `/api/user/{id}`
  - `/api/admin/users`

HOW TO PREVENT

- Do not rely on the client to enforce admin access.
- Deny all access by default.
- Only allow operations to users belonging to the appropriate group or role.
- Properly design and test authorization.

A6: MASS ASSIGNMNET

The API takes data that client provides and stores it without proper filtering for whitelisted properties. Attackers can try to guess object properties or provide additional object properties in their requests, read the documentation, or check out API endpoints for clues where to find the openings to modify properties they are not supposed to on the data objects stored in the backend.

USE CASES

- The API works with the data structures without proper filtering.
- Received payload is blindly transformed into an object and stored.
- Attackers can guess the fields by looking at the `GET` request data.

HOW TO PREVENT

- Do not automatically bind incoming data and internal objects.
- Explicitly define all the parameters and payloads you are expecting.
- Use the `readOnly` property set to `true` in object schemas for all properties that can be retrieved through APIs but should never be modified.
- Precisely define the schemas, types, and patterns you will accept in requests at design time and enforce them at runtime.

A7: SECURITY MISCONFIGURATION

Poor configuration of the API servers allows attackers to exploit them.

USE CASES

- Unpatched systems
- Unprotected files and directories
- Unhardened images
- Missing, outdated, or misconfigured TLS
- Exposed storage or server management panels
- Missing CORS policy or security headers
- Error messages with stack traces
- Unnecessary features enabled

HOW TO PREVENT

- Establish repeatable hardening and patching processes.
- Automate locating configuration flaws.
- Disable unnecessary features.
- Restrict administrative access.
- Define and enforce all outputs, including errors.

A8: INJECTION

Attackers construct API calls that include SQL, NoSQL, LDAP, OS, or other commands that the API or the backend behind it blindly executes.

USE CASES

- Attackers send malicious input to be forwarded to an internal interpreter:
  - SQL
  - NoSQL
  - LDAP
  - OS commands
  - XML parsers
  - Object-Relational Mapping (ORM)

HOW TO PREVENT

- Never trust your API consumers, even if they are internal.
- Strictly define all input data, such as schemas, types, and string patterns, and enforce them at runtime.
- Validate, filter, and sanitize all incoming data.
- Define, limit, and enforce API outputs to prevent data leaks.

A9: IMPROPER ASSETS MANAGEMENT

Attackers find non-production versions of the API (for example, staging, testing, beta, or earlier versions) that are not as well protected as the production API, and use those to launch their attacks.

USE CASES

- DevOps, the cloud, containers, and Kubernetes make having multiple deployments easy (for example, dev, test, branches, staging, old versions).
- Desire to maintain backward compatibility forces to leave old APIs running.
- Old or non-production versions are not properly maintained, but these endpoints still have access to production data.
- Once authenticated with one endpoint, attackers may switch to the other, production one.

HOW TO PREVENT

- Keep an up-to-date inventory all API hosts.
- Limit access to anything that should not be public.
- Limit access to production data, and segregate access to production and non-production data.
- Implement additional external controls, such as API firewalls.
- Properly retire old versions of APIs or backport security fixes to them.
- Implement strict authentication, redirects, CORS, and so forth.

A10: INSUFFICIENT LOGGING & MONITORING

Lack of proper logging, monitoring, and alerting allows attacks and attackers go unnoticed.

USE CASES

- Logs are not protected for integrity.
- Logs are not integrated into Security Information and Event Management (SIEM) systems.
- Logs and alerts are poorly designed.
- Companies rely on manual rather than automated systems.

HOW TO PREVENT

- Log failed attempts, denied access, input validation failures, or any failures in security policy checks.
- Ensure that logs are formatted so that other tools can consume them as well.
- Protect logs like highly sensitive information.
- Include enough detail to identify attackers.
- Avoid having sensitive data in logs — if you need the information for debugging purposes, redact it partially.
- Integrate with SIEMs and other dashboards, monitoring, and alerting tools.