OWASP API Security Top 10 and Beyond part2

Prepared by: Falilat Owolabi

Date: Dec 16th, 2025

This week, I reviewed the OWASP API Security Top 10 (2023), focusing on items 4–10 and related vulnerabilities. I covered unrestricted resource consumption, broken function-level authorization, business flow abuse, server-side request forgery, security misconfiguration, improper inventory management, and unsafe consumption of third-party APIs. I also reviewed injection vulnerabilities, insufficient logging and monitoring, and business logic flaws. Below is a summary of each vulnerability.

API4:2023 Unrestricted Consumption

Unrestricted Resource consumption occurs when the provider of an API does not put a safeguard to stop users from making an unlimited request of the API, This in turn will cause execution time out of the API which leads to denial of service for other users. The provider might have to incur unnecessary financial loss fixing the problem cause during the overly consumption of the API

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability Average	**Prevalence Widespread : Detectability Easy**	**Technical Moderate : Business Specific**
Exploitation requires simple API requests. Multiple concurrent requests can be performed from a single local computer or by using cloud computing resources. Most of the automated tools available are designed to cause DoS via high loads of traffic, impacting APIs' service rate.	It's common to find APIs that do not limit client interactions or resource consumption. Crafted API requests, such as those including parameters that control the number of resources to be returned and performing response status/time/length analysis should allow identification of the issue. The same is valid for batched operations. Although threat	Exploitation can lead to DoS due to resource starvation, but it can also lead to operational costs increase such as those related to the infrastructure due to higher CPU demand, increasing cloud storage needs, etc.

	agents don't have visibility over cost impact, this can be inferred based on service providers' (e.g. cloud provider) business/pricing model.	

OWASP Preventive Measures include:
The API provider should limit how many times a request can be made by a user within a specified timeframe, also notify the user how many times they can make the request and a countdown indicating how many requests they have left. Add proper server-side validation for query string and request body parameters.

A misconfiguration or missing of the following can lead to unrestricted Resource Consumption:
1) Execution time out
2) Maximum allocable memory
3) Maximum number of file descriptor
4) Maximum number of processes
5) Maximum upload file size
6) Number of operation to perform in single fie API client request
7) Number of record per page to return in a single API request
8) Third party service provider limit

API5:2023 Broken Function Level Authorization

Broken Function Level Authorization occurs when an API does not correctly enforce which functions each role (user, partner, admin, etc.) is allowed to call. Instead of only accessing functions meant for their own role, an attacker can invoke functions intended for other roles (often administrative), allowing them to alter or delete data, or perform actions on behalf of other users. While BOLA is about unauthorized access to data, BFLA is about unauthorized use of functionality (e.g., transferring other users' funds, managing accounts, or changing configurations).
cause during the overly consumption of the API

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability	**Prevalence common:**	**Technical Severe:**

Easy	Detectability Easy	Business Specific
Exploitation requires the attacker to send legitimate API calls to an API endpoint that they should not have access to as anonymous users or regular, non-privileged users. Exposed endpoints will be easily exploited.	Authorization checks for a function or resource are usually managed via configuration or code level. Implementing proper checks can be a confusing task since modern applications can contain many types of roles, groups, and complex user hierarchies (e.g. sub-users, or users with more than one role). It's easier to discover these flaws in APIs since APIs are more structured, and accessing different functions is more predictable.	Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack and may lead to data disclosure, data loss, or data corruption. Ultimately, it may lead to service disruption.

OWASP Preventive Measures (BFLA-focused)

- Implement a centralized, consistent authorization module that is invoked by every business function (endpoints and methods), not sprinkled ad hoc in the code.
- Enforce a default deny policy:
    - No function is callable unless explicitly mapped to specific roles/groups (whitelist approach).
- Review all API endpoints and HTTP methods against the application's business logic and group hierarchy:
    - Identify which roles are allowed to call which paths and methods (e.g. GET /{userId}/account/balance for owners, GET /admin/account/{userId} only for admins).
- Ensure that:
    - All administrative controllers inherit from a base controller that enforces strict role/group checks.
    - Any administrative function implemented inside "regular" controllers still performs explicit authorization checks based on user group and role.
- Include BFLA scenarios in testing:
    - Try calling admin/partner endpoints as normal users.

- Try changing HTTP methods (e.g. PUT → DELETE) on existing endpoints.
- Try using admin-style paths revealed by error messages or documentation with low-privileged tokens.

API6:2023 Unrestricted Access to Sensitive Business Flows

Unrestricted Access to Sensitive Business Flows occurs when an API provider does not protect critical business workflows (e.g., purchase, registration, stock checks) from automated or excessive use. Attackers can identify and automate these flows to obstruct legitimate users, deplete inventory, or manipulate business processes. This differs from general DoS in that it targets specific business logic rather than raw resource exhaustion.

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability Easy	**Prevalence widespread: Detectability Average**	**Technical moderate: Business Specific**
Exploitation usually involves understanding the business model backed by the API, finding sensitive business flows, and automating access to these flows, causing harm to the business.	Lack of a holistic view of the API in order to fully support business requirements tends to contribute to the prevalence of this issue. Attackers manually identify what resources (e.g. endpoints) are involved in the target workflow and how they work together. If mitigation mechanisms are already in place, attackers need to find a way to bypass them.	In general technical impact is not expected. Exploitation might hurt the business in different ways, for example: prevent legitimate users from purchasing a product, or lead to inflation in the internal economy of a game.

OWASP Preventive Measures:

Mitigation should be implemented in two layers:

1. **Business Layer**
   - Identify sensitive business flows that could harm the business if excessively or automatically used (e.g., purchase flows, stock reservations, account creation, promotional redemptions).

- Assess business risk for each flow and prioritize protection accordingly.

2. **Engineering Layer**

Choose appropriate protection mechanisms based on business risk:

Human Detection Mechanisms:

CAPTCHA or advanced biometric solutions (e.g., typing patterns, mouse movement analysis) to require human interaction at critical points in workflows.

Device and Pattern Analysis:

- **Device fingerprinting:** Deny service to unexpected client devices (e.g., headless browsers, automation tools) to increase the cost and complexity for attackers.
- **Non-human pattern detection**: Analyze user flow patterns to detect automation (e.g., user accessed "add to cart" and "complete purchase" in less than one second, or skipped normal browsing steps).
- **IP-based restrictions:** Consider blocking IP addresses from Tor exit nodes and well-known proxy services.

API-Specific Protections:

- Secure and limit access to machine-consumed APIs (developer APIs, B2B APIs) that are easier targets because they often lack human verification mechanisms.
- Implement workflow-level rate limiting (not just per-endpoint) to prevent rapid, repeated execution of entire business flows.
- Add delays or verification steps between critical workflow stages (e.g., require email confirmation before completing a purchase).

Implementation Note:

Simpler mechanisms (basic CAPTCHA, IP blocking) are easier to implement but easier to bypass.

More sophisticated mechanisms (biometric analysis, advanced pattern detection) are harder to implement but provide stronger protection against determined attackers.

API7:2023 Server Side Request Forgery

Server Side Request Forgery (SSRF) occurs when an API provider does not validate or restrict user-supplied URLs used to fetch remote resources. An attacker can control which URLs the server requests, potentially exposing private data, scanning internal networks, or achieving remote code execution. This vulnerability allows

attackers to leverage the target server to make requests to internal services, cloud metadata endpoints, or external systems that would otherwise be inaccessible. resource exhaustion.

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability Easy	**Prevalence widespread: Detectability Average**	**Technical moderate: Business Specific**
Exploitation requires the attacker to find an API endpoint that accesses a URI that's provided by the client. In general, basic SSRF (when the response is returned to the attacker), is easier to exploit than Blind SSRF in which the attacker has no feedback on whether or not the attack was successful.	Modern concepts in application development encourage developers to access URIs provided by the client. Lack of or improper validation of such URIs are common issues. Regular API requests and response analysis will be required to detect the issue. When the response is not returned (Blind SSRF) detecting the vulnerability requires more effort and creativity.	Successful exploitation might lead to internal services enumeration (e.g. port scanning), information disclosure, bypassing firewalls, or other security mechanisms. In some cases, it can lead to DoS or the server being used as a proxy to hide malicious activities.

OWASP Preventive Measures

1 Network isolation: Isolate the resource fetching mechanism in your network. These features are typically intended to retrieve remote resources, not internal ones. Use network segmentation to prevent the fetching service from accessing internal services.

2. **Implement allowlists (whitelist approach) whenever possible:**
   - **Remote origins:** Only allow users to download resources from expected, trusted sources (e.g., Google Drive, Gravatar, specific CDNs).
   - **URL schemes and ports:** Restrict to allowed schemes (e.g., https:// only) and specific ports (e.g., 80, 443).
   - **Accepted media types:** Limit to specific media types for given functionality (e.g., only image URLs for image processing endpoints).

3. **Disable HTTP redirections:** Prevent the server from following redirects, as they can bypass initial validation and lead to accessing unintended resources.

4. **Use well-tested URL parsers**: Avoid custom or flawed URL parsing logic. Use established, maintained URL parsing libraries to avoid issues caused by URL parsing inconsistencies (e.g., handling of @, #, encoded characters).
5. **Validate and sanitize all client-supplied input data**: Implement strict validation for any user-provided URLs, checking:

- URL format and structure
- Allowed schemes and ports
- Blocked internal IP ranges (RFC 1918: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16)
- Blocked localhost variants (localhost, 127.0.0.1, 0.0.0.0)
- Blocked cloud metadata endpoints (169.254.169.254 for AWS, Azure, GCP)

6.**Do not send raw responses to clients:** Avoid forwarding raw responses from fetched resources back to clients. Instead, process and sanitize responses before returning them, or return only necessary data to prevent information leakage.

API8:2023 Security Misconfiguration

Security Misconfiguration occurs when an API provider fails to properly configure security settings across the API stack (network, infrastructure, application). This catch-all category includes misconfigured headers, missing or weak transit encryption, default accounts, unnecessary HTTP methods, lack of input sanitization, and verbose error messages. When exploited, these misconfigurations can compromise the confidentiality, integrity, and availability of API data, ranging from information disclosure to full system compromise.

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability Easy	**Prevalence widespread: Detectability Easy**	**Technical severe: Business Specific**
Attackers will often attempt to find unpatched flaws, common endpoints, services running with insecure default configurations, or unprotected files and directories to gain	Security misconfiguration can happen at any level of the API stack, from the network level to the application level. Automated tools are available to detect and exploit misconfigurations	Security misconfigurations not only expose sensitive user data, but also system details that can lead to full server compromise.

unauthorized access or knowledge of the system. Most of this is public knowledge and exploits may be available..	such as unnecessary services or legacy options.	

OWASP Preventive Measures

The API lifecycle should include:

1. **Repeatable Hardening Process**
    - Establish a standardized, repeatable process for deploying properly locked-down environments
    - Enable fast and easy deployment of secure configurations
2. **Configuration Review and Updates**
    - Regularly review and update configurations across the entire API stack, including:
    - Orchestration files (Docker, Kubernetes, Terraform)
    - API components (application servers, frameworks, libraries)
    - Cloud services (S3 bucket permissions, IAM roles, security groups)
3. **Automated Continuous Assessment**
    - Implement automated processes to continuously assess the effectiveness of configurations and settings in all environments (development, staging, production)
4. **Specific Security Controls**

Transit Encryption:

- Ensure all API communications (client to API server and downstream/upstream components) use TLS encryption, regardless of whether the API is internal, private, or public-facing

HTTP Methods:

- Be specific about which HTTP verbs each API endpoint can be accessed by
- Disable all other HTTP verbs (e.g., HEAD, TRACE, OPTIONS) that aren't required

Browser-Based Client APIs:

- Implement a proper Cross-Origin Resource Sharing (CORS) policy
- Include applicable security headers (e.g., Content-Security-Policy, X-Frame-Options, X-Content-Type-Options)

Input Validation:

- Restrict incoming content types/data formats to those that meet business/functional requirements
- Implement proper input sanitization for file uploads and user-supplied data

Server Chain Uniformity:

- Ensure all servers in the HTTP server chain (load balancers, reverse/forward proxies, back-end servers) process incoming requests uniformly to avoid desync issues

Response Schema Enforcement:

Define and enforce all API response payload schemas, including error responses

API9:2023 Improper Inventory Management

Improper Inventory Management occurs when an API provider exposes non-production (development, staging, test, UAT, beta) or unsupported (deprecated, legacy) API versions that are not protected with the same security rigor as production. These versions often lack patches, use weaker security controls, and may be connected to production databases, making them gateways to other API vulnerabilities. When present, attackers can exploit outdated endpoints, known vulnerabilities, or weaker authentication to gain unauthorized access to sensitive data or administrative functions.

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability Easy	**Prevalence widespread: Detectability Average**	**Technical moderate: Business Specific**
Threat agents usually get unauthorized access through old API versions or endpoints left running unpatched and using weaker security requirements. In some cases exploits are available. Alternatively, they may get access to sensitive data through a 3rd party with whom there's no reason to share data with.	Outdated documentation makes it more difficult to find and/or fix vulnerabilities. Lack of assets inventory and retirement strategies leads to running unpatched systems, resulting in leakage of sensitive data. It's common to find unnecessarily exposed API hosts because of modern concepts like microservices, which make applications easy to deploy and	Attackers can gain access to sensitive data, or even take over the server. Sometimes different API versions/deployments are connected to the same database with real data. Threat agents may exploit deprecated endpoints available

	independent (e.g. cloud computing, K8S). Simple Google Dorking, DNS enumeration, or using specialized search engines for various types of servers (webcams, routers, servers, etc.) connected to the internet will be enough to discover targets.	in old API versions to get access to administrative functions or exploit known vulnerabilities..

OWASP Preventive Measures

1. **Comprehensive API Inventory**
    - Inventory all API hosts and document important aspects of each:
    - API environment (production, staging, test, development)
    - Network access (public, internal, partners)
    - API version (v1, v2, v3, etc.)
2. **Integrated Services Inventory**
    - Inventory integrated services and document:
    - Their role in the system
    - What data is exchanged (data flow)
    - Data sensitivity
3. **Complete API Documentation**
    - Document all aspects of your API:
    - Authentication mechanisms
    - Error handling and responses
    - Redirects
    - Rate limiting policies
    - Cross-Origin Resource Sharing (CORS) policy
    - All endpoints, including their parameters, requests, and responses
4. **Automated Documentation Generation**
    - Generate documentation automatically by adopting open standards (OpenAPI, Swagger)
    - Include documentation build in CI/CD pipeline to ensure it stays current with API changes
5. **Documentation Access Control**

- Make API documentation available only to those authorized to use the API
- Avoid exposing internal documentation publicly

6. **External Protection for All Versions**
    - Use external protection measures (API security firewalls, WAFs) for all exposed versions of your APIs, not just the current production version
    - Apply the same security rigor to deprecated and non-production versions
7. **Avoid Production Data in Non-Production**
    - Avoid using production data with non-production API deployments
    - If unavoidable, treat these endpoints with the same security as production (same authentication, rate limiting, monitoring)
8. **Version Retirement Strategy**
    - When newer versions include security improvements, perform risk analysis to decide mitigation actions for older versions:
    - Backport security improvements to older versions if possible without breaking API compatibility
    - Quickly retire older versions and force all clients to move to the latest version if backporting isn't feasible
    - Establish clear deprecation timelines and communication with API consumers

API10:2023 Unsafe Consumption of APIs

Unsafe Consumption of APIs occurs when an API consumer (application) trusts data from third-party APIs without applying the same security standards used for user input. This is a trust issue: data from third-party APIs should be treated as untrusted. If a third-party API provider is compromised, that insecure connection becomes an attack vector, potentially leading to complete compromise of organizations that consume data insecurely. This is the only item in the OWASP API Security Top 10 that focuses on API consumers rather than providers.

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability Easy	**Prevalence common: Detectability Average**	**Technical severe: Business Specific**

Exploiting this issue requires attackers to identify and potentially compromise other APIs/services the target API integrated with. Usually, this information is not publicly available or the integrated API/service is not easily exploitable.	Developers tend to trust and not verify the endpoints that interact with external or third-party APIs, relying on weaker security requirements such as those regarding transport security, authentication/authorization, and input validation and sanitization. Attackers need to identify services the target API integrates with (data sources) and, eventually, compromise them.	The impact varies according to what the target API does with pulled data. Successful exploitation may lead to sensitive information exposure to unauthorized actors, many kinds of injections, or denial of service.

OWASP Preventive Measures

1. **Assess Third-Party API Security Posture:**
    - When evaluating service providers, assess their API security posture before integrating
    - Review their security practices, compliance certifications, and vulnerability management processes
    - Consider their track record and security incident history
2. **Secure Communication Channels:**
    - Ensure all API interactions happen over a secure communication channel (TLS)
    - Never use unencrypted connections (HTTP) for third-party API communications
    - Validate TLS certificates to prevent MITM attacks
3. **Validate and Sanitize All Third-Party Data:**
    - Always validate and properly sanitize data received from integrated APIs before using it
    - Treat third-party API data with the same level of distrust as user-supplied input
    - Implement strict input validation
4. **Control Redirects:**
    - Maintain an allowlist of well-known locations that integrated APIs may redirect to

- Do not blindly follow redirects from third-party APIs
- Validate redirect destinations against the allowlist before following
- Reject redirects to unexpected or untrusted locations

Beyond Top 10!

Injection vulnerabilities occur when APIs do not properly filter or sanitize user input before passing it to supporting systems, allowing attackers to execute malicious commands. These flaws are very common and can be found in SQL, NoSQL, LDAP queries, OS commands, XML parsers, and ORM implementations. Protection requires keeping data separate from commands and queries through comprehensive input validation and sanitization, using parameterized interfaces, escaping special characters appropriately, and limiting data exposure. Injection attacks have been around for decades, so there are many standard security controls available, but they must be consistently applied across all API endpoints and input vectors. Verbose error messages, unexpected API behavior, and database error responses can all serve as indicators to attackers that injection vulnerabilities exist, making proper error handling and input validation critical for API security.

Insufficient Logging and Monitoring leaves API providers operating without visibility into attacks, allowing malicious activities to go undetected until it is too late. Logs reveal patterns in API usage, provide evidence of abuse, create audit trails for compliance, and help detect suspicious activities and anomalous user behavior. Protection requires comprehensive logging of security events (failed authentication, denied access, validation errors), proper log format and detail to identify malicious actors, protecting log integrity and confidentiality, continuous monitoring of infrastructure and API functioning, centralized log management using SIEM systems, and custom dashboards and alerts for early detection. Without these controls, attackers have ample time to fully compromise systems, making logging and monitoring essential for improving API performance and security.

Business Logic Vulnerabilities are intended features of an application that attackers can use maliciously by exploiting misplaced trust or assumptions not properly enforced. These vulnerabilities are unique to each application's business processes and are difficult to detect with automated tools, requiring specific knowledge of the application's functionality. Common examples include trusting partners not to expose APIs, assuming users won't intercept and alter requests, and

features with instructions but no technical validation. Protection requires using a threat modeling approach during design, reducing trust relationships in favor of technical controls, regular developer training, and external security testing through bug bounties or penetration testing. Since business logic vulnerabilities can have the most significant impact when strong technical controls are not in place, they require careful consideration of how features can be misused and implementation of proper server-side validation and authorization checks.

End Of Report!