Top 10 vulnerabilities

1. Broken Object Level Authorization

* What is it?

If a user finds an endpoint, say, a user number in a browser URL, and changes that number to another and the API returns information that the users should not have access to, then we are dealing with a broken object-level authorization. BOLA involves a user being able to directly access resources that they should not be able to access, using a user input functionality. The reason for the vulnerability existing as a result of simply changing an ID parameter is that the API does not check whether a user owns a resource before that user can do anything to it (such as modifying or deleting the resource).

* Where is it typically found?

These bugs are usually found in APIs which use an ID in the request. This can take the form of UserIDs, UUID, cookies, but also URLs and other forms.

* What to do?

Therefore, this can be mitigated by ensuring that an identifier is combined with a check that any user has permission to access this resource. This can be as straightforward as an if statement, or using middleware.Ensure that the API checks the authorization of each request against the appropriate access controls. This includes verifying the user's identity and permissions before granting access to sensitive resources. Use appropriate authentication mechanisms (e.g., tokens, OAuth, JWT) to validate user identities.

<https://www.youtube.com/watch?v=YciLnEY1AN0>

1. Broken User Authentication

* What is it?

User authentication is at the core of using APIs safely. It allows administrators to access the API and secured resources while preventing regular users from accessing these secured resources, as well as other users’ data. In more concrete terms, broken user authentication reflects when an API should have an authentication system but does not in practice, or that the implemented authentication fails in some cases, allowing attackers to impersonate an authenticated user. This includes lacking the implementation of captchas or cooldowns to prevent brute-forcing, not having multi-factor authentication methods (to address [credential stuffing](https://owasp.org/www-community/attacks/Credential_stuffing)) or allowing previously-leaked passwords, sending sensitive information through unencrypted channels and potentially in plain view (i.e. in URLs), or using weak encryption keys

* Where is it found?

One attack vector for API authorization is the ever-present API Key. Private keys are designed to be secret but may be found left in erroneous git commits. While API keys are designed for use in authentication capacities, there have been instances where they have been used in authorization roles. Therefore, if an API key can be found by a third party, they might be able to access resources that they are not supposed to. API keys can be found in different ways: “Google Dorking”, and looking for APIs that generate or use tokens without securing them.

* What to do?

The correct application and implementation of authentication mechanisms such as OAuth by reading the relevant documentation and setting up authentication systems to existing standards are essential.

1. Patch or update the authentication system
2. Strengthen password policies
3. Implement secure authentication protocols: Use industry-standard authentication protocols like OAuth, OpenID Connect, or SAML, which provide secure authentication and authorization mechanisms.
4. Enable multi-factor authentication (MFA): Implement MFA to add an extra layer of security. This can involve using additional factors such as SMS codes, email verification, authenticator apps, or hardware tokens.
5. Implement secure session management: Ensure that session management mechanisms, such as session tokens or cookies, are implemented securely. Use random, unique session identifiers, enforce session expiration, and securely transmit session data over encrypted channels (HTTPS).
6. Secure password recovery mechanisms:Utilize secure password reset tokens, employ rate limiting to prevent abuse, and validate the identity of users before allowing them to change passwords.

3) Excessive Data Exposure

* What is it?

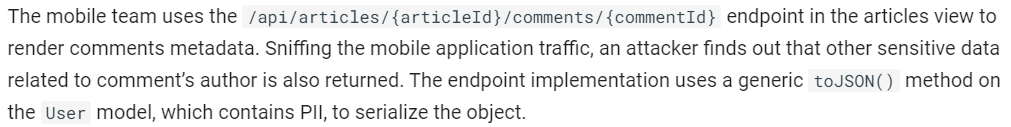
The API returns sensitive data to the client by design. This data is usually filtered on the client side before being presented to the user. An attacker can easily sniff the traffic and see the sensitive data.

This security vulnerability is simple in its functioning – too much information is passed on from the API to the client, with the client bearing the responsibility of filtering what API resources and other information are displayed to the end user. As a result, an API may return sensitive information, which although not displayed visually, still exists in the API. This is a security risk because, even when sensitive data may be designed to be accessible only by authorized and authenticated users, an attacker can view all the data sent by the API at the endpoint by circumventing the client or otherwise analyzing the traffic to the client from the API (man-in-the-middle attacks).

* What to do?

The first and foremost recommendation by OWASP to avoid excessive data exposure is to not rely on the client to carry out the information filtering, instead opting for the filtering to occur at the API level before the information is sent to the client. OWASP also suggests ensuring that each client receives only the information that is necessary. This step should be carried out at the API design stage, to ensure that once the API is deployed, a regular user’s client would only receive information that the client needs, and no extra data such as information that only administrators should have. Finally, they recommend classifying any sensitive and personal information and reviewing how this information is used by the API to avoid sending it when it is not necessary as part of the functionality.

Example Attack Scenarios



4) Lack of resources and rate limiting

API requests consume resources such as network, CPU, memory, and storage. The amount of resources required to satisfy a request greatly depends on the user input and endpoint business logic. Also, consider the fact that requests from multiple API clients compete for resources. An API is vulnerable if at least one of the following limits is missing or set inappropriately (e.g., too low/high):

* Execution timeouts
* Max allocable memory
* Number of file descriptors
* Number of processes
* Request payload size (e.g., uploads)
* Number of requests per client/resource
* Number of records per page to return in a single request-response

Quite often, APIs do not impose any restrictions on the size or number of resources that can be requested by the client/user. Not only can this impact the API server performance, leading to Denial of Service (DoS), but also leaves the door open to authentication flaws such as brute force.

An attacker uploads a large image by issuing a POST request to /api/v1/images. When the upload is complete, the API creates multiple thumbnails with different sizes. Due to the size of the uploaded image, available memory is exhausted during the creation of thumbnails and the API becomes unresponsive.

The exploitation mechanism for overloading an API with requests does not require authentication. It is also parallelizable and scalable so that either a single machine or several machines can be used concurrently to make requests to an API. Hardware on the API’s side can experience buffer overflows and exceptions but also run out of CPU, memory, network bandwidth, or disk space resources. It’s important to note that without rate-limiting for login functionalities, brute-forcing a password becomes possible. In this case, the lack of rate-limiting is a facilitator for another type of attack. As APIs with missing or improperly configured rate limits exist, denial-of-service attacks are particularly “easy” to carry out.

How To Prevent

Docker makes it easy to limit memory, CPU, number of restarts, file descriptors, and processes.

Implement a limit on how often a client can call the API within a defined timeframe.

Notify the client when the limit is exceeded by providing the limit number and the time at which the limit will be reset.

Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response.

Define and enforce maximum size of data on all incoming parameters and payloads such as maximum length for strings and maximum number of elements in arrays.

5) Broken Function Level Authorization

A broken function-level authorization essentially refers to a permission IDOR, whereby a regular user can carry out an administrator-level task. The broken function-level authorization refers to the user hierarchical permissions system being incomplete or broken. APIs that involve complex permissions and user roles that can span the hierarchy in different ways are more prone to having broken function-level authorizations.

The best way to find broken function level authorization issues is to perform deep analysis of the authorization mechanism, while keeping in mind the user hierarchy, different roles or groups in the application, and asking the following questions:

* Can a regular user access administrative endpoints?
* Can a user perform sensitive actions (e.g., creation, modification, or erasure) that they should not have access to by simply changing the HTTP method (e.g., from GET to DELETE)?
* Can a user from group X access a function that should be exposed only to users from group Y, by simply guessing the endpoint URL and parameters (e.g., /api/v1/users/export\_all)?

Don’t assume that an API endpoint is regular or administrative only based on the URL path.

While developers might choose to expose most of the administrative endpoints under a specific relative path, like api/admins, it’s very common to find these administrative endpoints under other relative paths together with regular endpoints, like api/users.

Scenario #1

During the registration process to an application that allows only invited users to join, the mobile application triggers an API call to GET /api/invites/{invite\_guid}. The response contains a JSON with details about the invite, including the user’s role and the user’s email.

An attacker duplicated the request and manipulated the HTTP method and endpoint to POST /api/invites/new. This endpoint should only be accessed by administrators using the admin console, which does not implement function level authorization checks.

The attacker exploits the issue and sends himself an invite to create an admin account:

POST /api/invites/new

{“email”:”hugo@malicious.com”,”role”:”admin”}

Scenario #2

An API contains an endpoint that should be exposed only to administrators - GET /api/admin/v1/users/all. This endpoint returns the details of all the users of the application and does not implement function-level authorization checks. An attacker who learned the API structure takes an educated guess and manages to access this endpoint, which exposes sensitive details of the users of the application.

How To Prevent

Your application should have a consistent and easy to analyze authorization module that is invoked from all your business functions. Frequently, such protection is provided by one or more components external to the application code.

The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.

Review your API endpoints against function level authorization flaws, while keeping in mind the business logic of the application and groups hierarchy.

Make sure that all of your administrative controllers inherit from an administrative abstract controller that implements authorization checks based on the user’s group/role.

Make sure that administrative functions inside a regular controller implements authorization checks based on the user’s group and role.

6) Mass Assignment

APIs contain information that only certain groups should be able to create, edit, update, or delete. Consider the outstanding customer debt in the case of a utility provider, for example – a payment system should be able to update this amount based on payments made by the customer, but the customer should not be able to directly edit this value. Mass assignment may make this value editable or deletable by the customer, which presents some potential for unwanted API behavior (since the utility company would not want their customers to decide how much they owe the company!). This issue is present at the API endpoint level.

How is this type of bug addressed?

Mass assignment endpoint vulnerabilities are found through reconnaissance. APIs that exhibit this behavior typically are built on frameworks; therefore, as part of reconnaissance, attackers often look for frameworks that may introduce this vulnerability by default. Another clue may be, for instance, when a user tries to change their password via an API, and the email field is automatically populated with the user’s email. These vulnerabilities are often born from a desire to make code reusable and abstract; however, this approach also inadvertently allows attackers to take advantage of these design decisions.

Whenever creating more flexible APIs, consider whether or not the client also requires this flexibility, or if sExample Attack Scenarios

Scenario #1

A ride sharing application provides a user the option to edit basic information for their profile. During this process, an API call is sent to PUT /api/v1/users/me with the following legitimate JSON object:

{"user\_name":"inons","age":24}

The request GET /api/v1/users/me includes an additional credit\_balance property:

{"user\_name":"inons","age":24,"credit\_balance":10}

The attacker replays the first request with the following payload:

{"user\_name":"attacker", "age":60, "credit\_balance":99999}

Since the endpoint is vulnerable to mass assignment, the attacker receives credits without paying.ome parameters are for internal use only.

1. Security Misconfiguration

Security-conscious developers may implement a myriad of security systems for their infrastructure and their APIs, however – as we’ve seen in some places in this list – a misconfiguration of security systems can be as dangerous as their complete absence. The reason for this is twofold: first, it allows malicious actors to bypass implemented systems, and second, it instils in developer and maintainer teams a false sense of security. It is therefore crucial that not only are the correct systems implemented but that they also are configured properly.

For Example: Inspecting traffic of a mobile application an attacker finds out that not all HTTP traffic is performed on a secure protocol (e.g., TLS). The attacker finds this to be true, specifically for the download of profile images. As user interaction is binary, despite the fact that API traffic is performed on a secure protocol, the attacker finds a pattern on API responses size, which he uses to track user preferences over the rendered content (e.g., profile images).

* To prevent exception traces and other valuable information from being sent back to attackers, if applicable, define and enforce all API response payload schemas including error responses.
* Ensure API can only be accessed by the specified HTTP verbs. All other HTTP verbs should be disabled (e.g. HEAD).
* APIs expecting to be accessed from browser-based clients (e.g., WebApp front-end) should implement a proper Cross-Origin Resource Sharing (CORS) policy.

1. Injection

Injection consists of sending an API malicious commands through a user input field, whether a text input, file upload, or other means. This attack vector allows malicious actors to send code or other executable commands to the API’s interpreter, which can be used to bypass security, change permissions, access information, damage, or disable the API. Common injections include cross-site scripting (XSS), SQL injections, and template injections. Injection flaws, such as SQL, NoSQL, Command Injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization.XSS refers to Cross-Site Scripting. This type of attack involves injecting a malicious script into a text field or other input field on a website, which typically contains code that is executed for another website user. These succeed commonly where there is no sanitation of user input data. Potential damage includes obtaining session data such as cookies that can then be used as part of another attack involving a broken object-level authorization, for instance.

For Example:

We have an application with basic CRUD functionality for operations with bookings. An attacker managed to identify that NoSQL injection might be possible through bookingId query string parameter in the delete booking request. This is how the request looks like: DELETE /api/bookings?bookingId=678.

The API server uses the following function to handle delete requests:

router.delete('/bookings', async function (req, res, next) {

try {

const deletedBooking = await Bookings.findOneAndRemove({'\_id' : req.query.bookingId});

res.status(200);

} catch (err) {

res.status(400).json({error: 'Unexpected error occured while processing a request'});

}

});

The attacker intercepted the request and changed bookingId query string parameter as shown below. In this case, the attacker managed to delete another user's booking:

DELETE /api/bookings?bookingId[$ne]=678

Preventing injection requires keeping data separate from commands and queries.

Perform data validation using a single, trustworthy, and actively maintained library.

Validate, filter, and sanitize all client-provided data, or other data coming from integrated systems.

Special characters should be escaped using the specific syntax for the target interpreter.

Prefer a safe API that provides a parameterized interface.

Always limit the number of returned records to prevent mass disclosure in case of injection.

Validate incoming data using sufficient filters to only allow valid values for each input parameter.

Define data types and strict patterns for all string parameters.

SQL INJECTION: <https://owasp.org/www-community/attacks/SQL_Injection>

You can test for injections using different tools. For example, ReadyAPI provides a paid tool for automatic scanning. Others, like Burp Suite, are partially free. Or, if you use Postman on a project, you could perform basic injection tests using Postman and data-driven testing.

1. Improper Assets Management

Improper assets management refers to when we have two versions of an API, v1 and v2, but we forget to delete v1 after starting to use v2, even if we fixed v1. The vulnerability here is that v1 might not use the latest security features, or involve obsolete features for which the documentation makes finding and fixing vulnerabilities difficult. Potential consequences of improper asset management include data leaks or server takeover through a common database between the current API (v2) and the old one (v1).Without the security features typically given through updates or upgrading to newer API versions with better support, an attacker can bypass security using known flaws. From the management side, a partial, incomplete, or outdated inventory of running instances of APIs can create the space for the old and unpatched APIs we just mentioned. The OWASP report also mentions that API hosts with unclear purpose, or without clearly documented answers to questions concerning the environment in which the API is running, the version, the information used and accessed, or who should have access to the API are certainly vulnerable. Other factors include APIs that don’t have a strategy in terms of life cycle and retirement.

The solution to improperly managed assets is, well, to properly manage them. This takes several forms. First and foremost, delete APIs which aren’t being used. There is no reason to keep them online and accessible either internally or externally if they are no longer in use. This avoids them being forgotten entirely (and consequently not updated nor given security patches), which may open them up to attack by malicious actors. The OWASP foundation also issued a variety of other recommendations concerning improper assets management. They include making an inventory of all used APIs, and including key information such as the current version, who can access the API, as well as the environment the API is in, and its current point in the life cycle. Also adding integrated services’ data flow and access, sensitivity, and role in the API system is important.

Concerning upgrading versions, OWASP makes an important note: “when newer versions of APIs include security improvements, perform a risk analysis to decide on the mitigation actions required for the older version: for example, whether it is possible to backport the improvements without breaking API compatibility or you need to take the older version out quickly and force all clients to move to the latest version.” Improper asset management can therefore be addressed by having an API rollout strategy with strong documentation and inventories to explain the purpose of an API, who can access it, and what is the API’s associated data flow, on top of applying security patches and updates regularly, and having a strategy that covers the entire design life cycle of the API, including retirement. These elements are all important to prevent malicious actors from exploiting improper management of API assets.

1. Insufficient Logging and Monitering

Logging and monitoring systems internal to a network and their communication with the outside world is essential in finding malicious actors who attempt to hack into a system. The same can be said about APIs: the monitoring and logging of changes, updates, information access by users, etc. ensure that any breaches are caught as fast as possible, and the mechanism through which the breach is accomplished can be defended against. Without logging and monitoring, IT and software development teams will not be aware of any discrepancies if (and when!) cyberattacks or hacking attempts occur. As a result, malicious actors can attack systems without being noticed, allowing them to cause lots of damage over time. Unfortunately, insufficient logging and monitoring are typically remedied after significant damage has been done, whether by leak, damage, or loss of information, or the damage done to a company’s reputation.

The API is vulnerable if:

1. It does not produce any logs, the logging level is not set correctly, or log messages do not include enough detail.
2. Log integrity is not guaranteed (e.g., Log Injection).
3. Logs are not continuously monitored.
4. API infrastructure is not continuously monitored.

For Example: Access keys of an administrative API were leaked on a public repository. The repository owner was notified by email about the potential leak, but took more than 48 hours to act upon the incident, and access keys exposure may have allowed access to sensitive data. Due to insufficient logging, the company is not able to assess what data was accessed by malicious actors.