frontegg

GUIDE

The Complete Guide to API Security

Table of content

Introduction

The Application Programming Interface (API) has revolutionized the SaaS space, allowing developers to focus on what matters most - innovation. However, these productivity-enhancing elements can become a security liability since they handle critical and sensitive information. With over 90% of apps testing positive for some kind of broken access control issue, securing APIs has now become a priority.

It's enough to take a look at the latest OWASP Top-10, where broken access control has taken the first spot, overtaking the evergreen injection vulnerabilities.

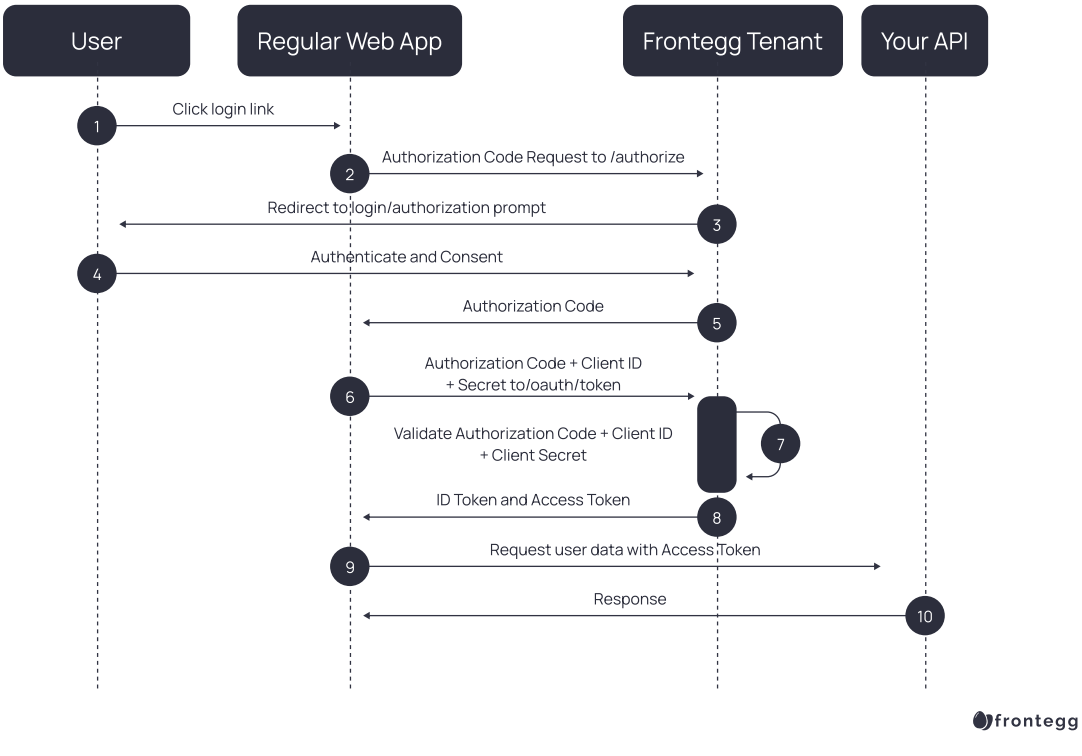
But how does one get started with API security? What's the right way to tackle this challenge? We have put together this comprehensive guide to help you get started on the right foot. Besides introducing the key components of the API ecosystem, you'll also get access to some useful best practices that will help you secure your application. Without further ado, let's dive into the thick of things.

The Access Token: OAuth vs Client Credentials vs Single Token Flow

The key component when it comes to API usage is the access token, which helps ensure secure communication between your apps, which has to be retrieved to gain access to the service. There are many ways to retrieve an access token. These ways are called flows or grants. OAuth supports various types of flows. The best-suited flow depends on your use case and application type - a key security requirement.

Whenever you click on the “Purchase” button on your favorite website to buy a Christmas gift, your order instantly gets placed. While it might feel like it happened instantaneously to you, a lot of things happen behind the scenes just to verify that you are indeed the user that is allowed to be performing this action. Let’s take a closer look at the three options and what they exactly do.

OAUTH AUTHORIZATION FLOW

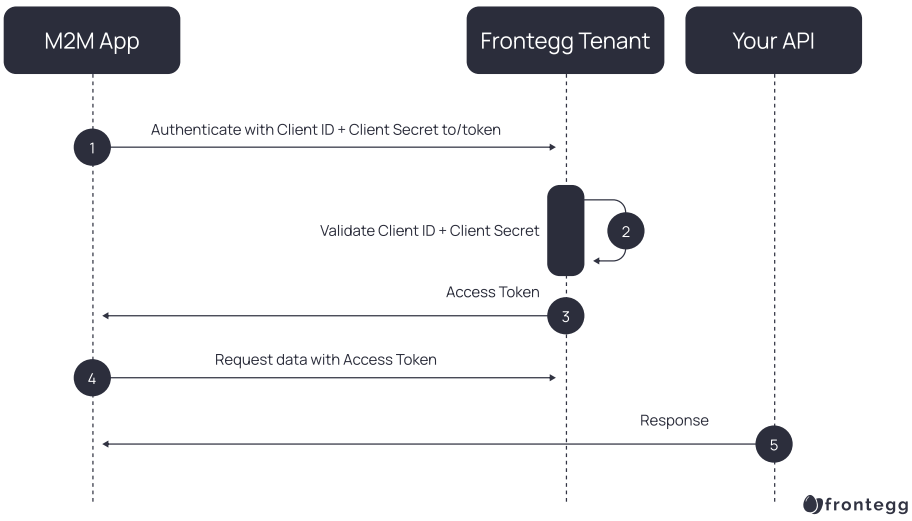


In a nutshell, the OAuth authorization flow helps exchange authorization codes for tokens, assuming your application is server-side. The application's Client Secret is also passed along and keeping it secure is top priority.

The flow looks as follows:

- The user clicks on the "Login" button in the regular web app
- The user is redirected to Frontegg's authorization server
- Frontegg's authorization server then redirects the user to the login and authorization prompt
- The user enter their credentials to login to the application and authorize it
- The Frontegg authorization server redirects the user back to the web application with an authorization code (one-time only)
- The authorization code is sent to the Frontegg authorization server, along with the Client Secret and the app's Client ID
- All of the mentioned items in the previous step are verified and validated
- The Frontegg authorization server sends a response with ID and Access Tokens, sometimes with a Refresh Token as well
- The app can now use the Access Token to call APIs and access user information
- The API responds with the data that has been requested.

CLIENT CREDENTIALS FLOW



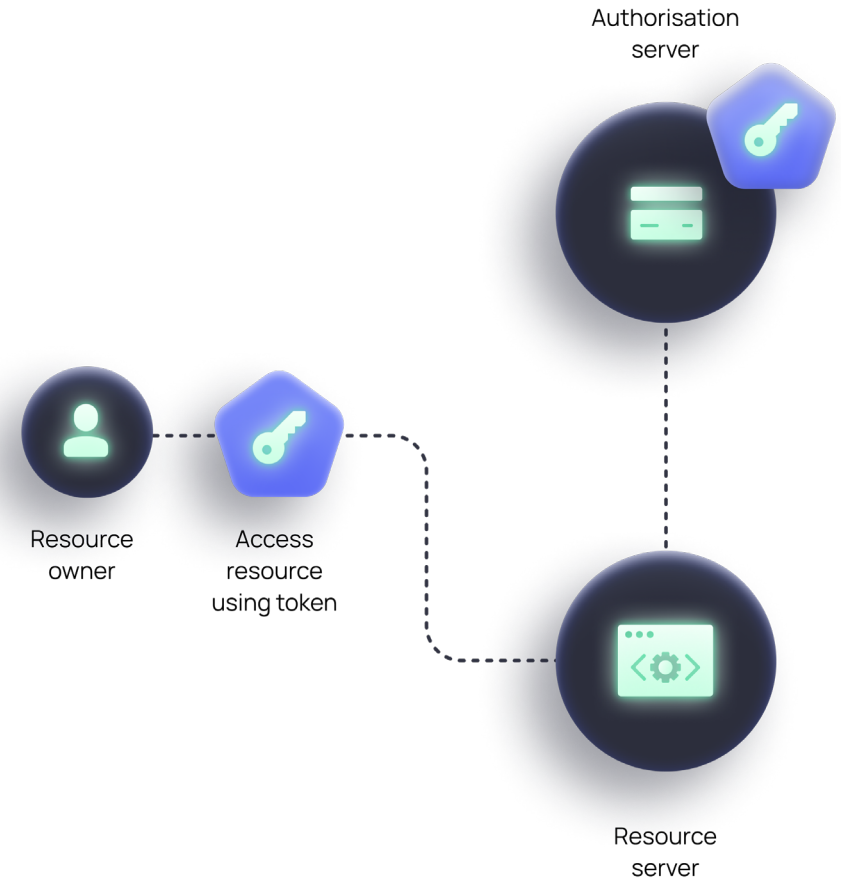
The Client Credentials Flow allows applications to pass their Client Secret and Client ID to an authorization server, which authenticates the user, and returns a token, without any user intervention. These flows are very relevant for machine-t0-machine (M2M) apps like deamons, back-end services, and CLIs, where the system authenticates and grants permissions without involving the users.

This is how the flow looks like:

- 1.The application authenticates with the OAuth authorization server, passing the Client Secret and Client ID
- 2.The authorization server checks the Client Secret and Client ID
- The Access Token is returned to the application
- The Access Token allows the application to access the target API with the required user account
- The API promptly responds with the requested data

SINGLE TOKEN FLOW

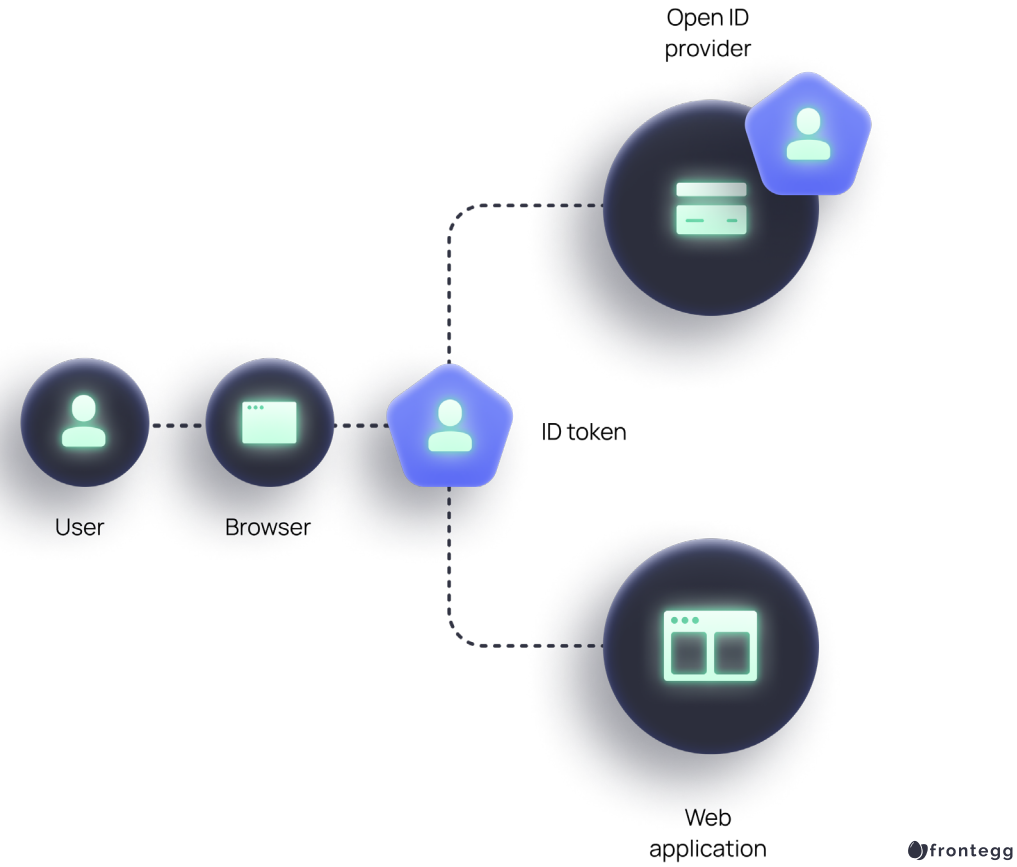
In a single token flow, we authenticate using a single token instead of generating a Client ID and Client Secret. This is to simplify the authentication flow and provide clients with a token that does not need to pass through the authorization server. There are multiple ways to do this. You can either generate a JWT token and leave out the expiration field or generate a client token to be validated against a central token store.



The Token Types: ID Token vs Access Token vs Personal Token

There are various types of tokens when it comes to securing your API. We'll be looking into ID tokens and access tokens in the context of Frontegg.

ID TOKEN



An ID token works as an identifier to indicate that a user has been authenticated. An ID token is encoded in JWT (JSON Web Token), which is a standard format for making the inspection of tokens and data easier for the applications.

A typical encoded JWT Token looks like the following.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwOi8vbXktZG9tYWluLmF1dGgwLmNvbSIsInN1YiI6ImF1
```

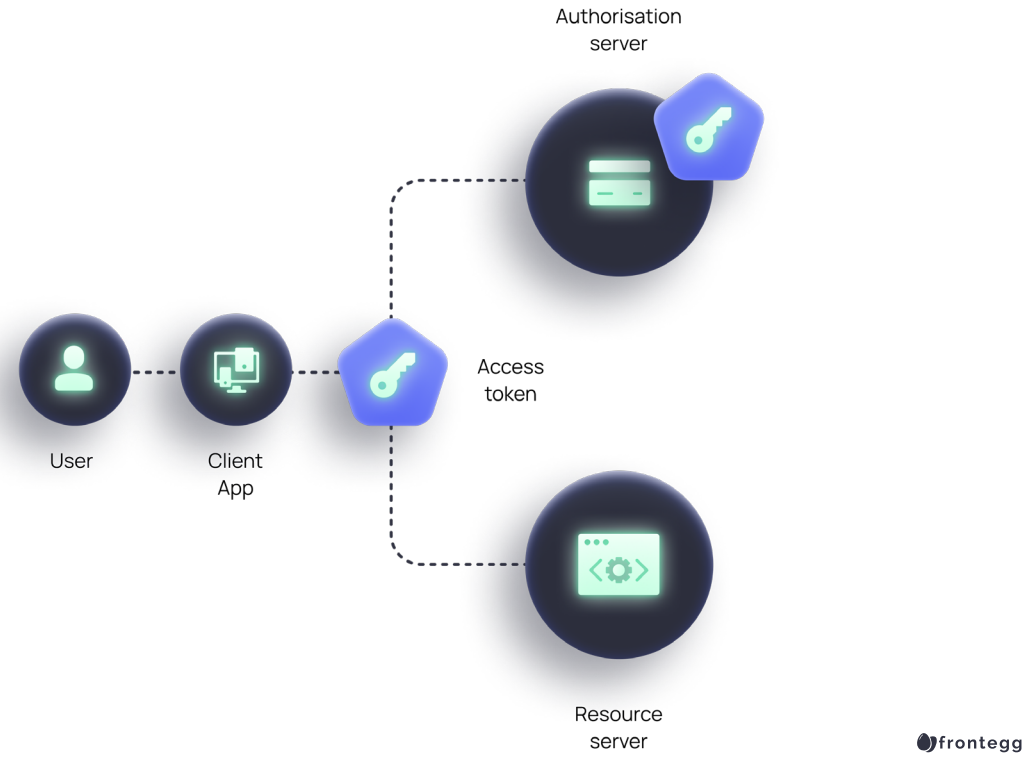
This is just the base64 encoded form of the JSON data. It will look like the following if we base64 decode the JWT Token.

```
{
  "iss": "http://my-domain.auth0.com",
  "sub": "auth0|123456",
  "aud": "1234abcdef",
  "exp": 1311281970,
  "iat": 1311280970,
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe"
}
```

Here, one of the most important properties is the “aud” property. It defines the final recipient of the token. The ID token may have additional information about the user, such as their email address, picture, birthday, and so on.

Finally, perhaps the most important thing—the ID token is signed by the issuer with their private key. This guarantees you the origin of the token and ensures that it has not been tampered with, which can be verified using the issuer’s public key.

ACCESS TOKEN



The access token is the artifact that allows the client application to access the user's resource. It is issued by the authorization server after it successfully authenticates the user and obtains their consent. The access token allows a client application to access a specific resource to perform specific actions on behalf of the user. It can be a string in any format. A common format used for access tokens is JWT.

PERSONAL TOKEN

Personal access tokens are alternatives to using your passwords for authenticating with a service. For example, you can use the web version of GitHub and use your username and password to authenticate. However, if you are working with the command-line version of GitHub, then using user credentials for authentication becomes infeasible.

That's where personal access tokens come into play. Usually, personal authentication tokens are removed from the server if not used for an extended period. For example, GitHub uses a personal token system to authenticate its users. But, if a key is not used in over a year, then GitHub automatically removes it from the user account.

An In-Depth Look at the Token Repository

Now that we know how important tokens are in terms of API security and have learned about the various types of tokens, we can dive into token generation.

If we use OAuth Flow, we need to generate a Client Secret and ID to save in the database for future use. The Client ID and Secret will be generated as 128 UUID and version 4. We use the user password, hash it, and salt it to generate the Client Secret. This makes it harder for attackers to determine passwords in the case of a database hijack. The Client Secret and Client ID is stored in the database in such a way that it is easy to find the token for each user.

After considering all the design choices, our final JWT token looks like this:

Link to code

```
export interface UserApiToken {
  clientId: string;    // The client ID
  secret: string;      // This will be stored as hash
  userId: string;      // This will be the user ID which this token belongs to
  scopes: [string];    // This will be the scopes relevant for the token
  createdAt: Date;     // This will be the time when the token was created
}
```

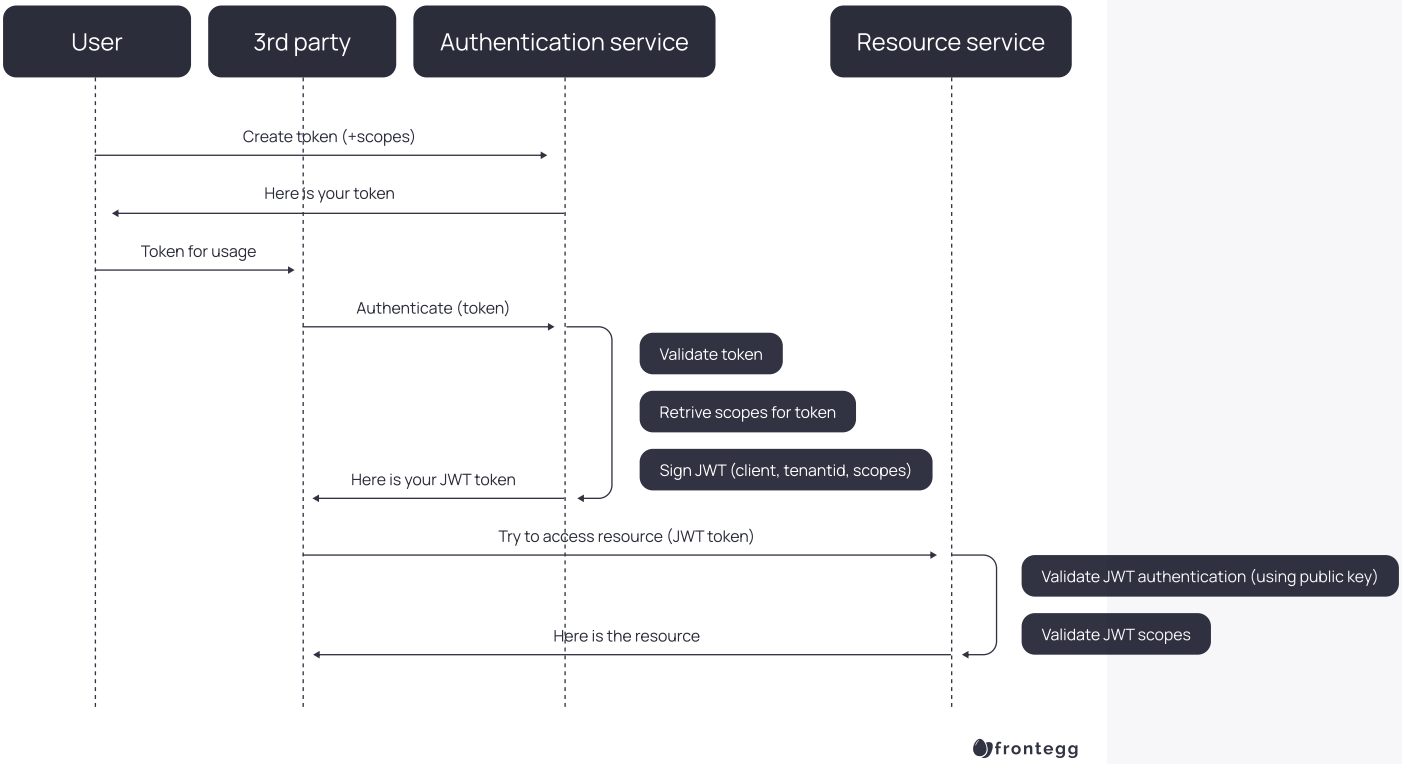
Here, we are storing the Client ID and Client Secrets. Additionally, we're also storing the user ID to maintain a relationship between the token and the user account. It also contains important information such as scopes relevant to the token and the time when the token was created.

Once a token is created, it is passed to other apps. As we do not want the 3rd party service to modify the scopes on the token, API tokens are generated as immutable objects. Even token creators are not allowed to change scopes/ ClientID/Secret, etc.

Authorization with API Tokens

Now that we have seen the general design of API tokens, we will dive into authorization flow with API tokens. API tokens are not used for authentication only. We also need to maintain a list of permissions on the signed JWT to verify each microservice call.

A simple flow looks like this:



First, a user sends the request to the authentication service to create a token. The authentication service processes the user request and creates the JWT token to send back to the user. The user then combines the token with their request and sends it to a 3rd party website for authentication. To verify the user's identity, the 3rd party also performs various checks with the authentication service.

A 3rd party website sends the authentication request with the token to the authentication service. The authentication service now validates the token, retrieves the scope for that token, and finally signs the JWT token to send it to a 3rd party website. Now, 3rd party websites can request the resource to a resource service using a JWT token.

When implemented in code, it looks something like this:

```
import { verify } from 'jsonwebtoken';

export function validateAuthenticationAndScopes({ scopes = [] }) {
  return async (req, res, next) => {
    const authorizationHeader: string = req.header('authorization');
    if (!authorizationHeader) {
      return res.status(401).send('Unauthenticated');
    }

    const token = authorizationHeader.replace('Bearer ', '');

    // Verify the JWT token signature using the public key
    verify(token, publicKey, { algorithms: ['RS256'] }, (err, decoded: any) => {
      if (err) {
        res.status(401).send('Authentication failed');
        return next(err);
      }

      if (scopes && scopes.length > 0) {
        for (const requiredScope of scopes) {
          if (!decoded.scopes.includes(requiredScope)) {
            res.status(403).send('Insufficient scopes');
            return next('Insufficient scopes');
          }
        }
      }

      // And move to the next handler
      next();
    });
  };
}
```

Link to code

Tracking Token API Activity: Audit Logs for Compliance and Scrutiny

As security is the main focus, it is essential to have traceability on our API tokens. This means we need to audit every authentication and every API token activity. There are a few things to keep in mind when you are implementing traceability for API tokens. It is called the 5 Ws checklist.

- The Who – It should be possible to identify which user created the API token. So, if some API token has a security compromise attached to it, we can link that incident to a user.
- The When – It should be possible to identify the time an API token was created. This helps triage a security incident and identify the specific time range when a security compromise occurred.
- The What – What scopes are attached to this token? Knowing this helps one identify permissions associated with the API token.
- The Where – Important user data, such as from where this token was accessed, should be logged in the database.
- The How – How was that token used? The request method, context, and last authenticated time should be logged.

> Injection vulnerabilities have been knocked off the top of the latest OWASP Top-10 rankings by broken access control flaws. API security has to be taken seriously today.

FRONTEGG
Aviad Mizrahi
co-founder and CTO

API Security: 10 Best Practices

While this list of best practices is not exhaustive by any way or form, it is a great way to get started and improve your security posture. Let's get started.

1 – IMPLEMENT PROPER TOKEN VERIFICATION

If the API backend is verifying the token but is not checking if the token is associated with the object that is being requested, it can result in a case of broken object-level authorization. In this vulnerability, an attacker obtains access to an object by breaching the security of other objects or data streams.

In most cases, the reason behind broken access control is improper permission assignments. To implement it properly, various authorization and verification checks should be performed using the id assigned from the session, instead of the one given by the user.

2 – CHANGE DEFAULT CREDENTIALS

It is not uncommon for API developers to use default credentials on APIs to authenticate API users, which leads to broken authentication and security compromises.

3 – USE RANDOM ACCESS TOKENS

One way to secure yourself against broken authentication is to use random access tokens and short-lived tokens while rate-limiting API requests. Multi--factor authentication should also be employed to ensure security even when credentials are compromised.

4 – ENSURE THERE IS NO EXCESSIVE DATA EXPOSURE

In various cases, APIs return more data than required by the application. A potential attacker can take advantage of this excess information to attack the API service. To prevent these bugs, one should never rely on client applications for data filtering, and always ensure use cases before sending any personally identifiable information in response.

5 – IMPLEMENT PROPER RESOURCE AND RATE LIMITING

If not handled properly, many APIs do not check requests made by API users. For every request, there is some cost and overhead attached. If attackers send multiple requests until the backend is out of resources, then the API service's availability will be compromised. These attacks are more commonly called denial of service attacks.

6 – IMPLEMENT PROPER FUNCTION LEVEL AUTHENTICATION

It is really common for API developers not to implement function-level authorization. This results in an attacker carrying out tasks that are not permitted. Consider the scenario of a deleteUser endpoint on a query. Only authorized administrators should be able to call that query. But, if anyone can call the endpoint, then it would be a huge security risk.

7 – MASS ASSIGNMENT PREVENTION

In cases where an API is used to modify some values of an object, input is directly processed to modify values without any checks. For example, the following call is made by a user to change the account name.

```
GET /changeAccountName?name=Bob
```

But malicious actors can specify additional parameters to compromise the integrity of the object. A malicious request would look like this::

```
GET /changeAccountName?name=Bob&AccountBalance=999
```

To prevent this, all fields that are not supposed to be changed by the user should be set to read-only.

8 – PERFORM SECURITY MISCONFIGURATION CHECKS

Attackers frequently come up with vulnerabilities in applications and software components. If your backend components and applications are vulnerable, it can lead to significant data breaches. To prevent this, one should keep all backend infrastructures and services updated with all patches and perform regular security audits.

9 – PREVENTION AGAINST INJECTION ATTACKS

IInjection attacks are one of the biggest threats to applications working with web components. Injection attacks occur if the input from the user is completely trusted and directly processed by the backend without any input filter. This can result in a data compromise or even a complete web service takeover.

10 – IMPLEMENT SUFFICIENT LOGGING AND MONITORING

It is common for many large-scale deployments to set up infrastructures for logging and monitoring. If this is designed poorly and the logs are not properly monitored with security information and event management (SIEM) systems, it becomes a huge issue when a security compromise occurs.

API Security is Just The First Step

APIs are everywhere and the possibility of getting hacked is real. While the task of securing your API may seem daunting, you can focus on your business without worrying about security by implementing the tips provided in this guide. But API security is just one aspect of comprehensive API management, which needs to be taken seriously to achieve maximum security and optimize performance metrics.

You will also need to properly govern your APIs to give devs proper access to documentation, while also providing them with the required authentication and authorization capabilities. Besides governance, there is also a need for real-time data to gain insights into usage patterns and promote analytical development. APIs are great to have in any ecosystem, but only an end-to-end management infrastructure can help you stay safe and secure your data as you scale up.