End-to-End API Security

A guide to securing your web service APIs

WSO2

Table of Content

1

Introduction

APIs are a cornerstone technology for organizations seeking to digitally transform their business. Exposing corporate data and custom-built functionality through a standardized set of web service endpoints allows companies to enhance the value of strategic partnerships, rapidly develop new digital products, and personalize experiences for their customers. As the world relies more heavily on digital connections, many leadership teams are relying even more on their API programs to respond to rapidly changing customer and market demands.

This reliance on API strategies is reflected in the explosion of tools to help manage and control access to those services. For instance, the global API management market is expected to grow at a compound annual growth rate (CAGR) of 32.9% to reach USD 5.1 billion by 2023 according to MarketsandMarkets.

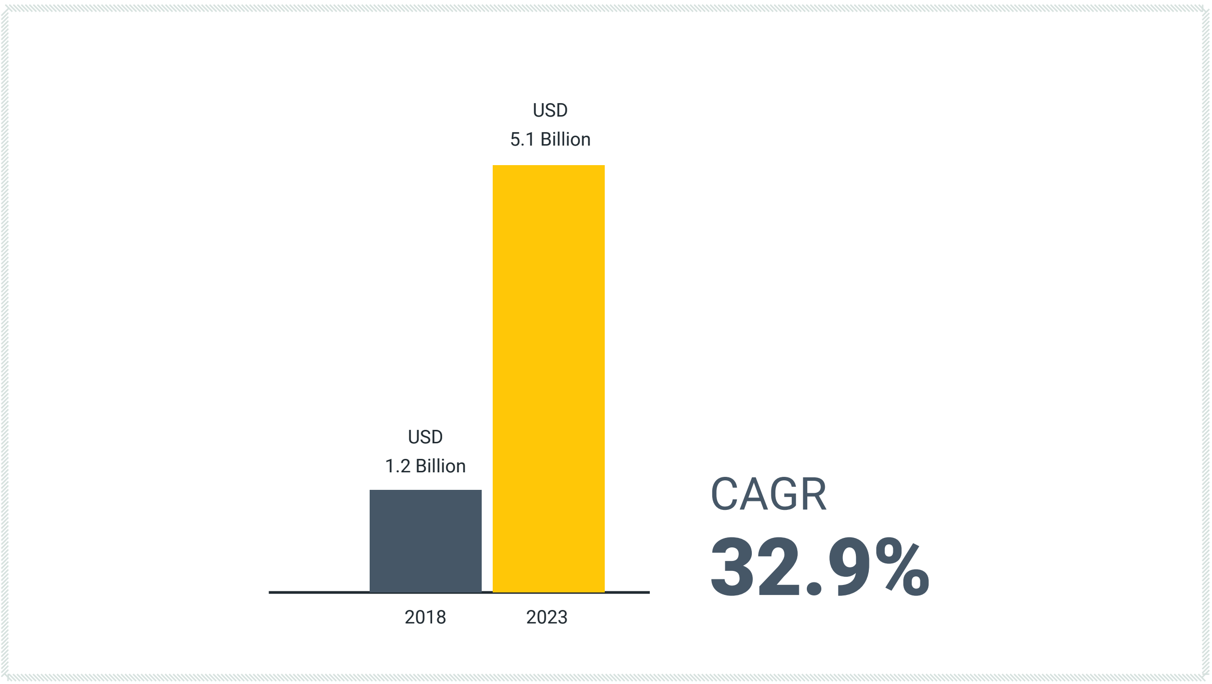


Figure 1: The growing API management market

As APIs become more critical to business success, the focus must shift to ensuring their reliability and security. In this book, we will explore these security challenges in detail and guide you through the process of addressing them.

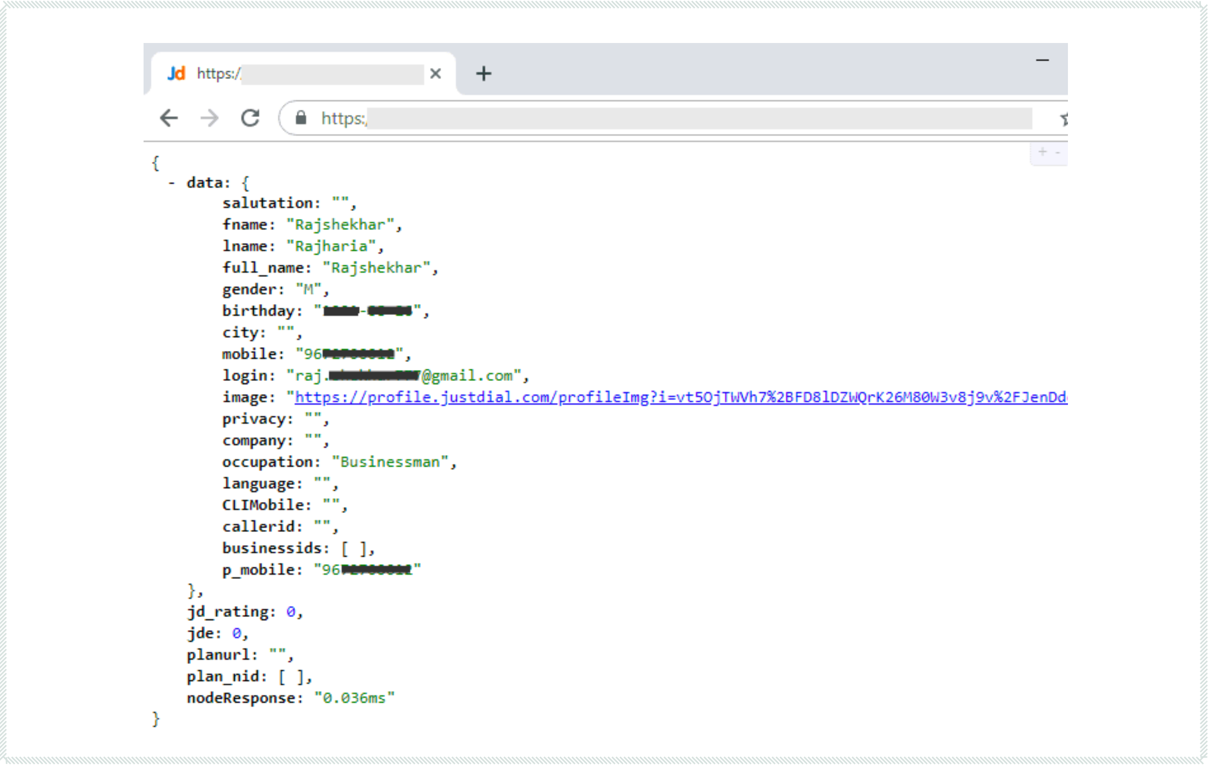
Confronting the Security Challenges of APIs

API security discussions are often limited to the public APIs an organization exposes to build an application ecosystem or drive strategic integrations. A strong set of API security best practices has emerged over the years to address these use cases, and products such as WSO2's API Manager were developed to make it easy to tightly control access to web services, manage their traffic, and monitor their use.

According to Gartner[1], however, most APIs created today are built for internal audiences. In most cases, developers employed by the organization use these APIs to leverage privileged data to create new digital products that serve the goals of the business. The end-users of these applications are often customers and others external to the organization, accessing the company's APIs indirectly, usually through a browser-based or mobile interface. Although these APIs are not intended for use by external developers, they still expose a potential attack surface for bad actors to exploit if not properly secured.

As API-first approaches to application development and integration have gained wide adoption, a few high-profile security breaches have made headlines:

- In 2018, an uncaught vulnerability in Facebook's Developer API exposed the personal data of nearly 50 million users.
- In 2019, a data breach at JustDial placed the data of more than 100 million users at risk due to an unprotected publicly accessible API endpoint.
- Later that same year, a developer discovered an API key in a public GitHub repository that would have allowed anyone to access a critical part of Starbucks' access control systems.



```
{
  - data: {
        salutation: "",
        fname: "Rajshekhar",
        lname: "Rajharia",
        full_name: "Rajshekhar",
        gender: "M",
        birthday: "[redacted]",
        city: "",
        mobile: "96[redacted]",
        login: "raj.[redacted]@gmail.com",
        image: "https://profile.justdial.com/profileImg?i=vt5OjTWVh7%2BFD8lDZWQrK26M80W3v8j9v%2FJenDd
        privacy: "",
        company: "",
        occupation: "Businessman",
        language: "",
        CLIMobile: "",
        callerid: "",
        businessids: [ ],
        p_mobile: "96[redacted]"
    },
    jd_rating: 0,
    jde: 0,
    planurl: "",
    plan_nid: [ ],
    nodeResponse: "0.036ms"
}
```

Figure 2: Data breach at JustDial

These incidents have been an eye-opener for most business organizations. A majority of respondents to a 2021 Salt Security survey said they had delayed the launch of an application due to API security concerns. Of those same respondents, 91 percent also reported having uncovered an API vulnerability in the previous year, indicating their security concerns are also born from experience.

Placing APIs behind a gateway that provides strong authentication and authorization controls, restricts traffic based on predefined rules, and delivers real-time usage statistics is a crucial first step in securing APIs that can significantly reduce vulnerability exposure with little effort. However, in a world where just about everything digital is delivered through an API, tracking where the data is flowing outside of the API system itself becomes more critical. Recall that the aforementioned Starbucks vulnerability was not an attack on the APIs themselves, but the leak of a secret authentication key buried in a publicly accessible codebase.

Businesses must do more than merely secure their web services — they must adopt a culture that looks at potential vulnerabilities at every stage of development, from planning through deployment. As APIs gain wider adoption, they will continue to evolve in their usage. Attackers are also evolving in their tactics. It is incumbent on digital businesses to stay vigilant and take responsibility for the security of their and their customers' data.

Through our focus in this document is on securing the web services themselves, we encourage the diligent reader to research secure development processes and practices to ensure privileged data is not able to leak from their systems in any form. Topics such as "zero trust", "automated vulnerability scans", and "secrets management" will provide a solid jumping-off point.

API Security Considerations

To ensure API security, organizations should pay close attention to the following areas. They will also help consumers in keeping their data safe and interacting with the correct services.

- Access control
- API protection
- Threat detection
- Issue analysis (ability to track a fraudulent activity)

API architects and security champions should focus on all these areas when designing their APIs. These areas can be further discussed using the topics below.

3.1 The API gateway pattern

When talking about API security, first we must understand where it should be addressed. Typically, an API is created by a set of API developers to perform a set of business operations using specified business logic. If all the security aspects are also included within the API itself, the implementation logic of the API becomes more complex.

In addition, a particular API can be accessed by a wide range of devices and applications (in most cases, these applications require different security mechanisms). Therefore, it is important to comprehensively address API security concerns and prevent misuse.

To address these requirements, the best practice is to offload API security responsibilities to an API gateway. The API gateway resides between the API backend and consumers (clients). It will intercept all the requests that are initiated by consumers and manage the security-related aspects. That way, API developers (and developers) can focus on the business logic and API functions, while the security and access control part is managed by the API gateway. This also gives the ability to extend security capabilities to address a wide range of access requests, which come from different client applications and devices.

3.2 API access control

Access control to APIs is about governing who can access which APIs and about what functionalities of those APIs they are allowed to consume. This is covered by the Authentication and Authorization protocols of APIs.

3.2.1 Authentication

Authentication, in simple terms, is about identifying the entity (user) requesting access to the API, and (based on that identity) deciding whether or not to grant access. Identification is typically performed by either validating something the user knows (a password) or something the user has (a card, fingerprint, etc.). In somewhat advanced cases, a combination of these is used, such as verifying a password and verifying the correctness of a one-time password (OTP) sent to the user via SMS.

There are many variations of authentication. Password-based, token-based, certificate-based, and bio-metric-based authentication are some examples.

3.2.1.1 Basic authentication

This is the simplest authentication method used in an HTTP authentication process. User credentials (a username and password) are encoded using the base64 algorithm and it is attached to an HTTP header when sending a request. There is a dedicated HTTP header to send this encoded credential. This will be picked up by the authorization mechanism before serving the request.

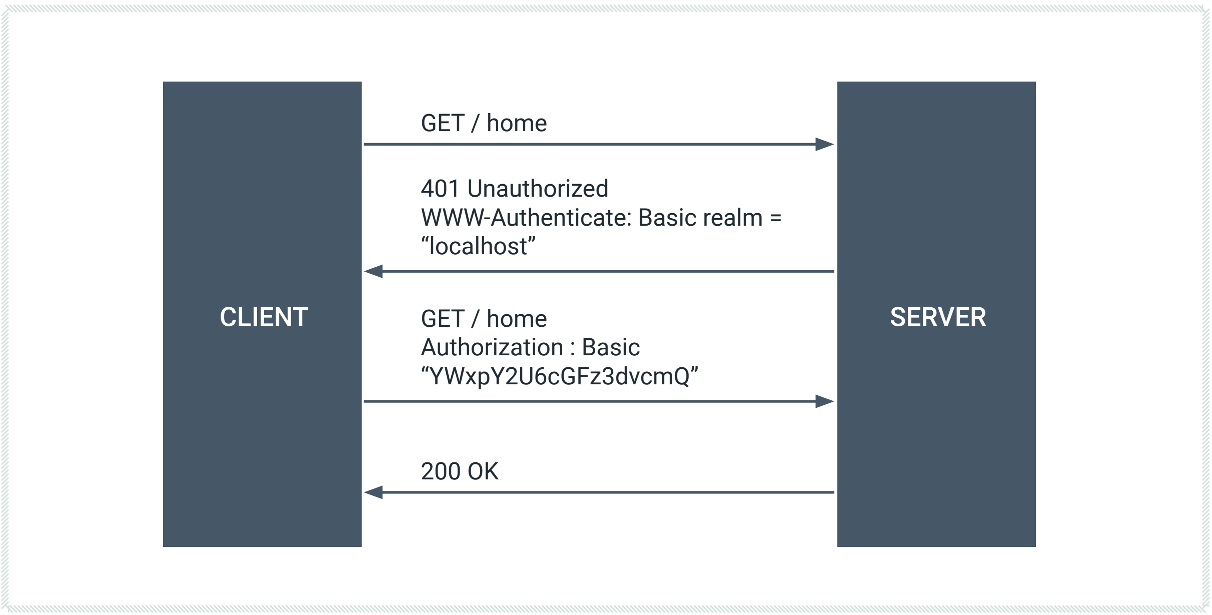

CLIENT
SERVER
GET / home
401 Unauthorized
WWW-Authenticate: Basic realm = "localhost"
GET / home
Authorization : Basic "YWxpY2U6cGFz3dvcmQ"
200 OK

Figure 3: Basic authentication

However, basic authentication is not sufficient to protect APIs against all complex security attacks, and it has some significant limitations. The username and password have to be shared with at least two parties. If a particular API is accessed by various types of client applications, the credentials have to be shared between all of them. There is another threat associated with this method. If the credentials are stolen by a third party, they can use those credentials to access the secured services without even getting noticed by the actual user. This threat will be there as long as the password is modified. However, by this time, attackers may have accessed secure information (or services) and they might abuse them or expose them in the future.

Due to the concerns mentioned above, using basic authentication has become a less secure approach (and it is not recommended for public APIs). OAuth was introduced to address these vulnerabilities. In OAuth, the credentials (username/password) are not shared directly. Instead, an access token is used. This access token is associated with a specific application. Moreover, this token has a lifetime. This means even that token cannot be used perpetually. This reduces the threat of token theft because even if it is stolen, it cannot be used for a long period of time.

In addition, the OAuth token can also be associated with scopes. Here, the token can only be used to access specified (or protected) resources based on roles assigned to a user. This way, more control can be established for role-based access control. This minimizes the risk of both intentional and unintentional misuse of a token because, with a given token, only a limited number of actions can be performed. Hence, OAuth is recommended for securing APIs because it is much safer than a password.

3.2.1.2 OAuth 2.0

Over the years, OAuth 2.0 has become the de-facto authorization mechanism for APIs. Using OAuth2.0, a user is authorized to use an access token. The token has a limited lifetime and is authorized to perform a limited set of functions (scopes) on the resource server.

OAuth 2.0 is an open standard for access delegation. This is mainly used as a way for internet users to grant websites or applications access to their information on other websites. Here, an access token is issued based on a client ID and a client secret associated with the application.

In OAuth 2.0, as per the specification, four roles can be identified

- Resource Owner
- Resource Server
- Client Application
- Authorization Server

Here is a brief description of each role.

Resource Owner

The resource owner is the person or application that owns the data that is to be shared. For instance, a user of a social media account could be a resource owner. The resource they own is their data. The resource owner is depicted in the diagram as a person, which is probably the most common situation. The resource owner could also be an application. The OAuth 2.0 specification mentions both possibilities.

Resource Server

The resource server is the server hosting the resources. Concerning our example, the social media network is a resource server (or has a resource server).

Client Application

The client application is the application requesting access to the resources stored on the resource server (which are owned by the resource owner). For example, a client application could be a game requesting access to a user's social media account.

Authorization Server

The authorization server is the server authorizing the client (application) to access the resources of the resource owner. Sometimes, the authorization server and the resource server can be the same server, but it doesn't have to be. If they are separate, communication between these two servers has to be designed by developers of the resource server and authorization server.

A detailed description of OAuth 2.0 can be found on this website for further reading.

Now, it is time to explore the different ways of obtaining a token. This is where OAuth grant types come into the picture.

OAuth 2.0 Grant Types

When it comes to OAuth 2.0-based authentication, there are several key components. Those are the resource owner, client, authorization server, and resource server.

Depending on the relationship between each of these parties, the way to acquire an access token (which represents a user's permission for the client to access data) might differ. This is where grant types come into the picture. Moreover, these grant types make the OAuth 2.0 specification a flexible authorization framework.

Here is a brief description of each grant type and when to use them.

Authorization Code Grant

This is optimized for confidential clients. Here, it is possible to authenticate the client and transmit the access token directly to the client without passing it through the resource owner's user-agent and potentially expose it to others (including the resource owner).

Since this is a redirection-based flow, the client should be capable of interacting with the resource owner's user-agent and receive incoming requests (via redirection) from the authorization server.

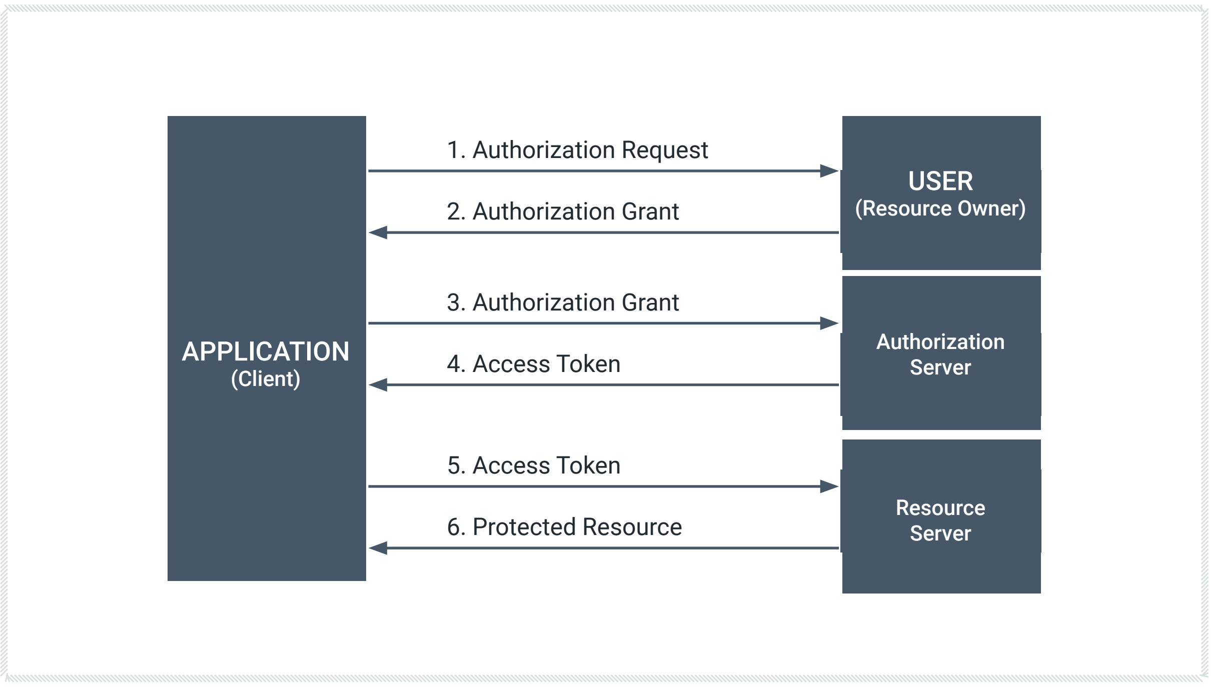


Figure 4: Authorization code flow

Password Grant Type

This grant type is used when the resource owner has a trusted relationship with the client (e.g., the device operating system or a highly privileged application). The client should be able to obtain the credentials of the resource owner. In this case, the authorization server has a critical responsibility. It should only allow this grant type if other grant types flows are not viable.

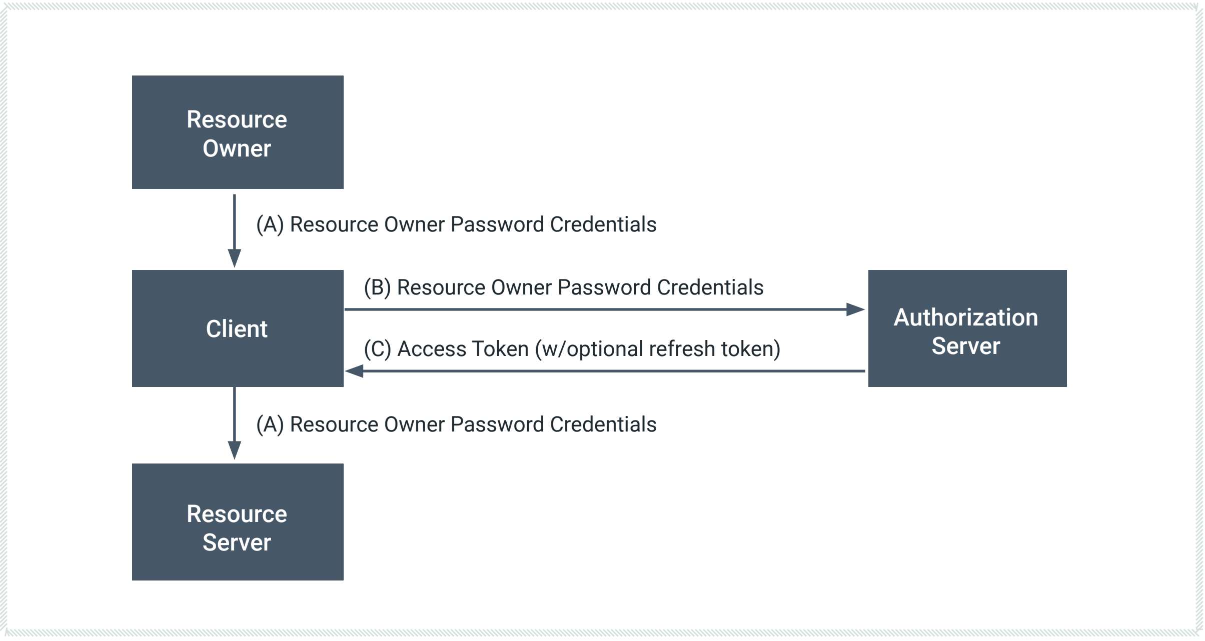

Resource Owner
(A) Resource Owner Password Credentials
Client
(B) Resource Owner Password Credentials
(C) Access Token (w/optional refresh token)
Authorization Server
(A) Resource Owner Password Credentials
Resource Server

Figure 5: Password grant

Client Credentials Grant Type

This grant type is not associated with a resource owner. Here, an application is needed and there are two values associated with it — the client ID and the client secret. By using those credentials, an access token will be issued with a specified lifetime. In most cases, this grant type is used when the application owner and the resource owner are the same.

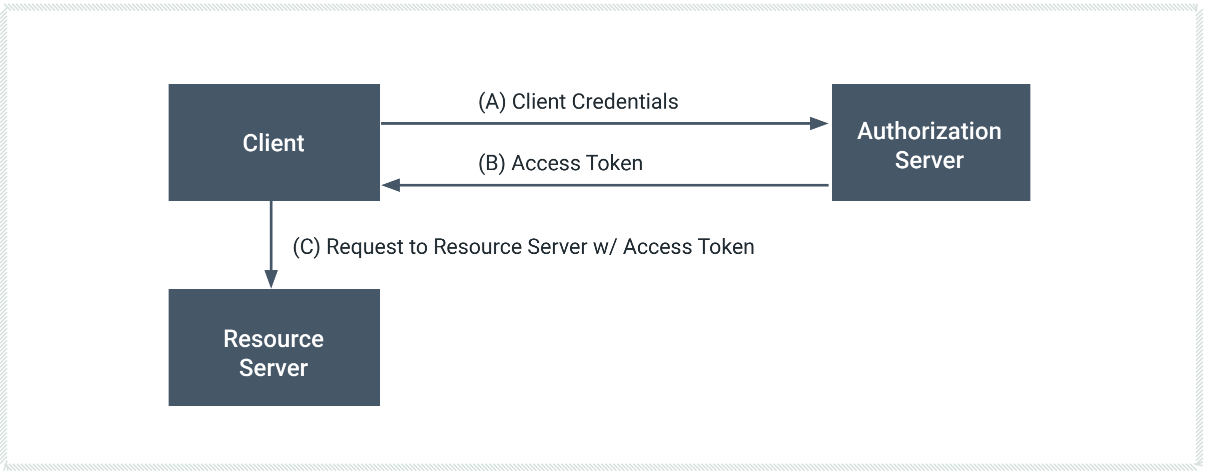


Figure 6: Client Credential Grant

Kerberos OAuth2 Grant Type

The Kerberos grant type supports a wide array of operating systems and applications. Most popular browsers also support this grant type. A Kerberos ticket can be exchanged for an OAuth 2.0 token. This allows organizations to use their existing Kerberos infrastructure and still adopt OAuth 2.0.

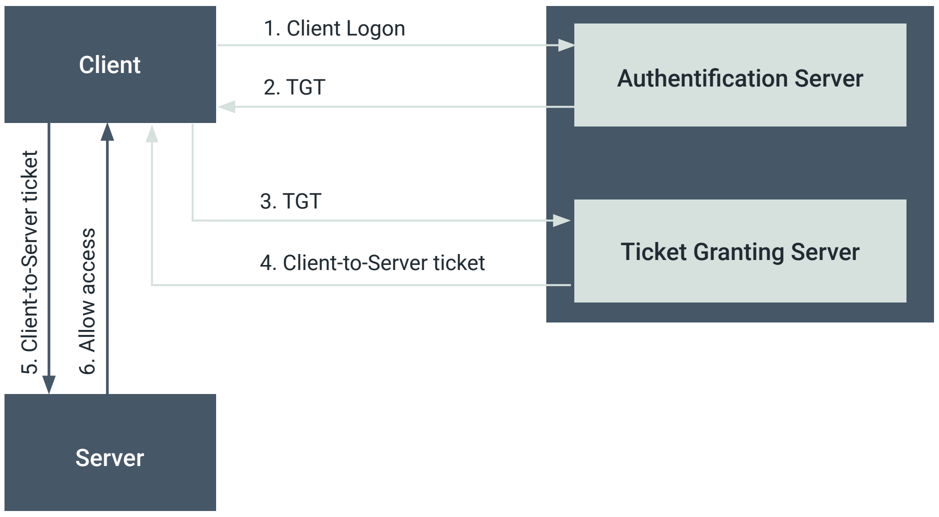


Figure 7: Kerberos grant type

Refresh Token Grant Type

Usually, an access token is valid for a limited time. It is not recommended to have access tokens with limitless lifetimes, because if a token is acquired by an attacker, it can be used permanently without being noticed. In this situation, it is necessary to renew the token when the lifetime expires. This is where refresh tokens come into the picture. Users can send the refresh token (issued along with the access token) and obtain a new access token to use the service continuously.

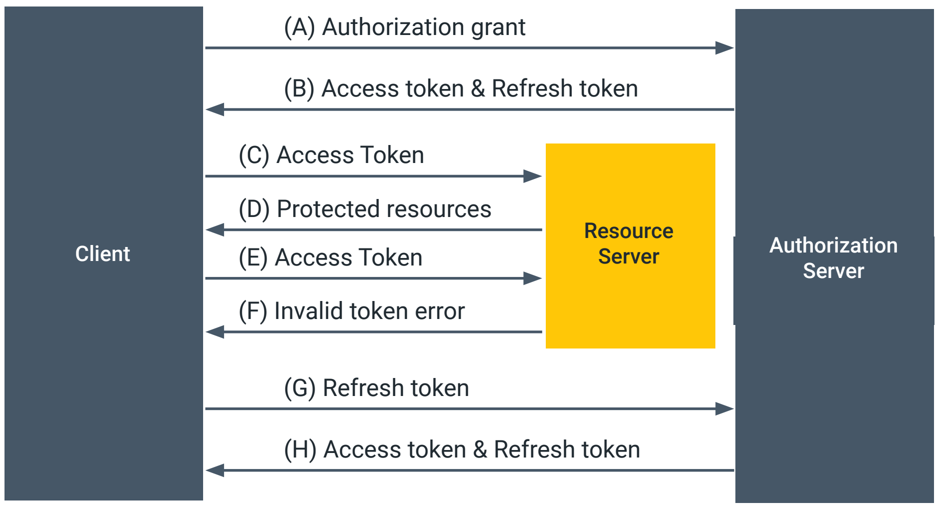


Figure 8: Refresh token grant

JWT Grant Type

This is an open security standard to securely communicate data between multiple parties as JSON objects. It defines a compact and self-contained way to securely transmit information between parties; verified and trusted because it is digitally signed. In this grant type, it is possible to send user claims to the backend to validate. Before issuing an access token, the authorization server can use these claims to properly identify the end-user.

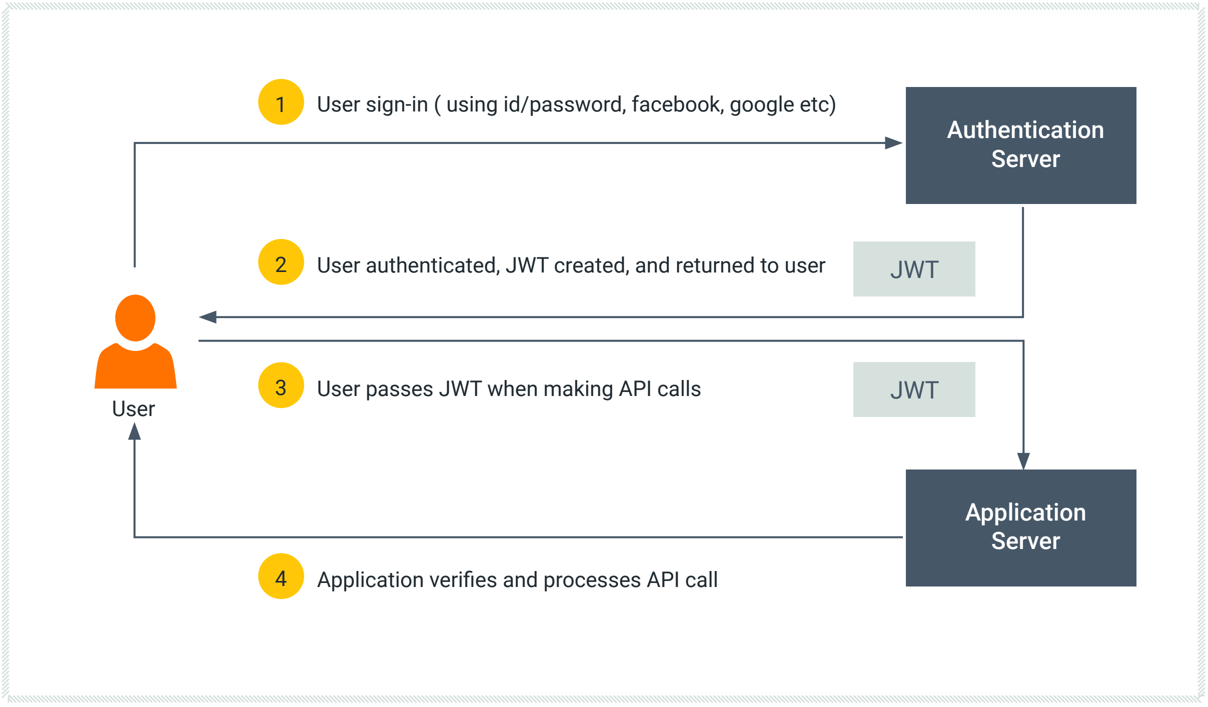

1 User sign-in (using id/password, facebook, google etc)
Authentication Server
2 User authenticated, JWT created, and returned to user
JWT
User
3 User passes JWT when making API calls
JWT
Application Server
4 Application verifies and processes API call

Figure 9: JWT Flow

A detailed technical description of each grant type can be found in the OAuth specification for further reading.

3.2.1.3 OIDC-based authentication

OIDC is referred to as OpenID Connect. This is built on top of the OAuth 2.0 protocol. In this authentication mechanism, it is possible to verify the identity of the end-user based on the authentication performed by an authorization server, while obtaining some profile details about the user using the REST-like mechanism. The standard for OIDC is governed by the OpenID foundation.

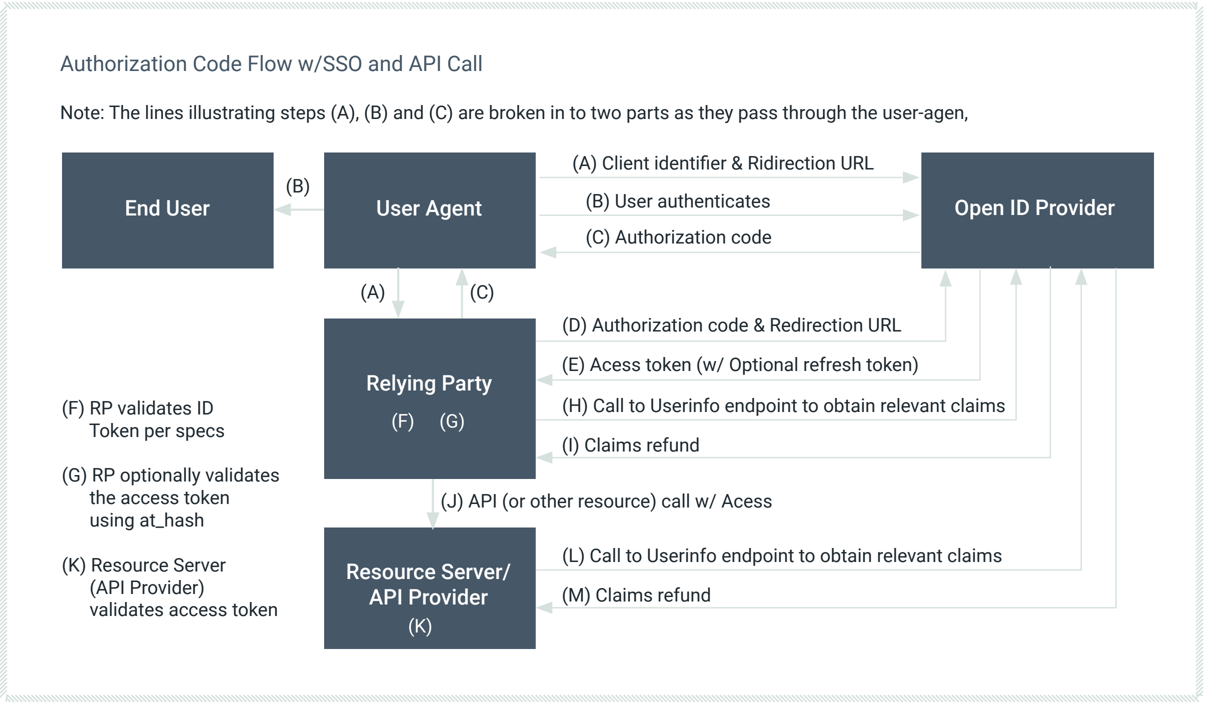

Authorization Code Flow w/SSO and API Call
Note: The lines illustrating steps (A), (B) and (C) are broken in to two parts as they pass through the user-agen,
End User
User Agent
Open ID Provider
(A) Client identifier & Ridirection URL
(B) User authenticates
(C) Authorization code
(B)
(A)
(C)
Relying Party
(F) (G)
(D) Authorization code & Redirection URL
(E) Acess token (w/ Optional refresh token)
(H) Call to Userinfo endpoint to obtain relevant claims
(I) Claims refund
(J) API (or other resource) call w/ Acess
Resource Server/ API Provider (K)
(L) Call to Userinfo endpoint to obtain relevant claims
(M) Claims refund
(F) RP validates ID Token per specs
(G) RP optionally validates the access token using at_hash
(K) Resource Server (API Provider) validates access token

Figure 10: OIDC flow

3.2.1.4 API key-based authentication

This authentication mechanism is a form of application-based security. This is a simple form of app-based security and it can be easily configured for an API. The API key is simply a string value passed by a client app to the APIM gateway. This is used to uniquely identify the client app, and if the app is valid, the flow continues.
Note that API key-based authentication is most suitable for internal APIs within the organization.

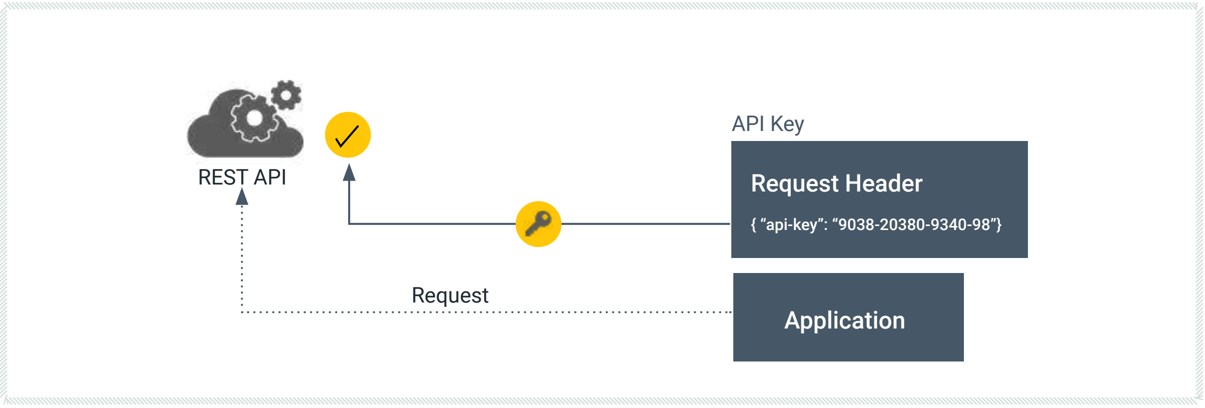


Figure 11: API key-based authentication

3.2.1.5 Mutual SSL authentication

This is a certificate-based authentication where two parties authenticate each other by verifying the provided digital certificate so that both parties are assured of the others' identity. In other words, a client authenticates themselves to a server and that server also authenticates itself to the client. This process is implemented using digital certificates issued by the trusted certificate authorities (CA). The certificates are managed using a set of public keys and private keys so that both parties can rely on each other for a secure connection. In most cases, mutual SSL authentication is suitable for APIs used by partners or known groups of clients.

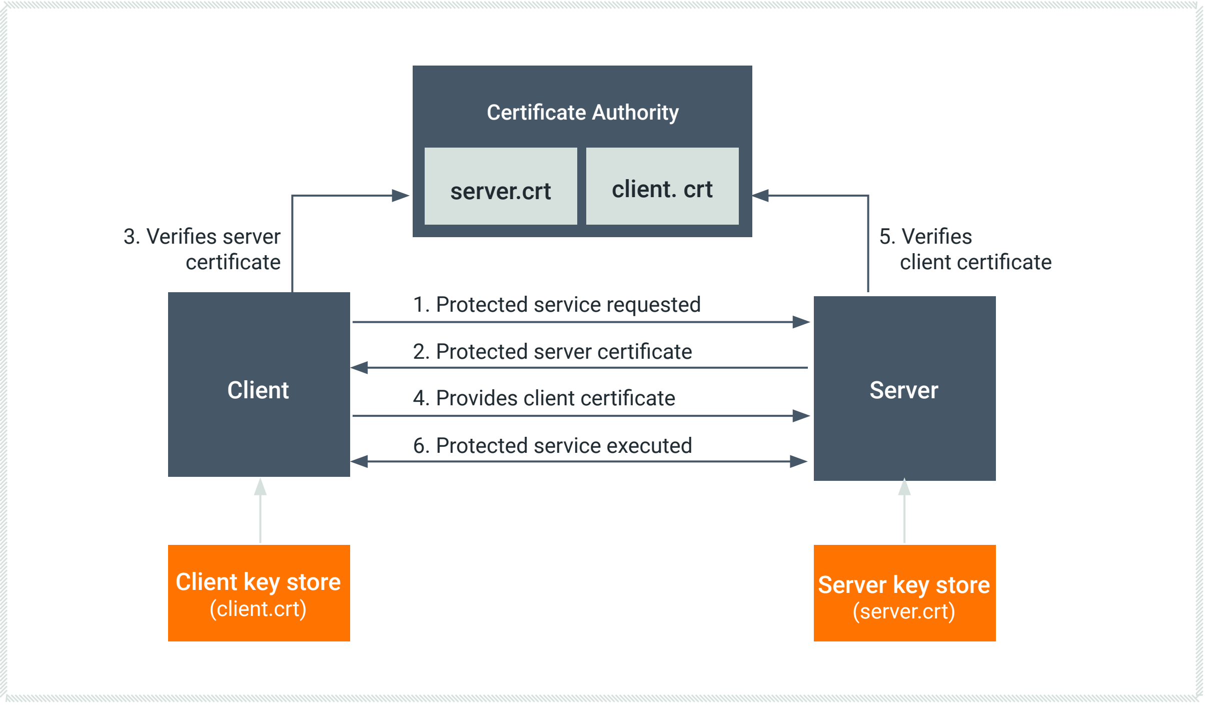

Certificate Authority
server.crt
client. crt
3. Verifies server certificate
5. Verifies client certificate
1. Protected service requested
2. Protected server certificate
4. Provides client certificate
6. Protected service executed
Client
Server
Client key store (client.crt)
Server key store (server.crt)

Figure 12: Mutual SSL

3.2.2 Authorization

The purpose of the authorization is to determine access levels or user privileges related to system resources. These system resources can be resources in APIs, files, data, and even features of the system.

3.2.2.1 Role-based access control

OAuth 2.0 has become the most widely used protocol for API authorization. OAuth 2.0 access tokens can be used to consume any resource of allowed APIs. However, there may be cases where it is necessary to implement more fine-grained access control. Scopes become a useful solution for this situation.

A scope enables fine-grained access control for each resource of the API based on specific consumer roles (role-based access control). Scopes ensure that only the allowed resources can be accessed by the consumer. If the resource is not allowed for consumption (based on the role of the consumer), then consuming that resource will not be allowed even though there is a valid access token.

There are many examples of this. A person with the “manager” role can perform create, read, and edit operations while a person with the “employee” role is only allowed to access the read operations. Both users are using the same API (with valid tokens), but they can only consume the allowed operations.

API	ShoppingCartAPI		
Resource	Items (/items)		
Action	PUT (Edit operation)	GET (Read operation)	POS (Create operation)
Roles	manager	manager, employee	manager

3.2.2.2 XACML-based access control

External extensible Access Control Markup Language (XACML) is an XML-based, declarative access control policy language based on XML. It can provide a standardized way of validating authorization requests. This can be used to have fine-grained, role-based access control for APIs. An access control policy language, request/response language, and reference architecture is defined in XACML. In the policy language, it defines access control policies (i.e., who is allowed to do what operations at what time). Queries about whether a particular access should be allowed (requests) and answers to those queries (responses) are defined in the request/response language. Finally, the reference architecture proposes a standard for the deployment of necessary software modules within an infrastructure to allow efficient enforcement of policies.

Attribute-Based Access Control (ABAC) and evaluation are supported by XACML. These evaluations are done with the additional data retrieved from the Policy Information Point (PIP). This PIP is defined in the XACML reference architecture.

In this context, access rights are granted to consumers by using policies that combine attributes. Any type of attributes such as user attributes, resource attributes, object attributes, or environment attributes can be useful in these policies.

Here is a representation of the XACML architecture.

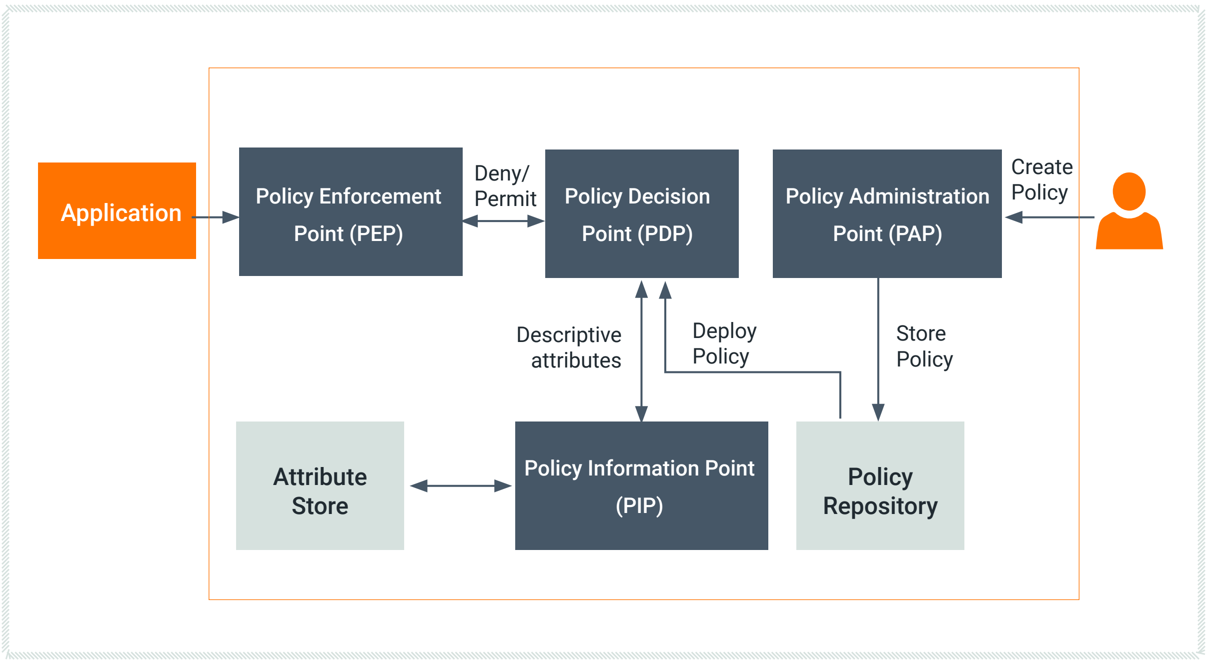

Application
Policy Enforcement Point (PEP)
Deny/ Permit
Policy Decision Point (PDP)
Policy Administration Point (PAP)
Create Policy
Descriptive attributes
Deploy Policy
Store Policy
Attribute Store
Policy Information Point (PIP)
Policy Repository

Figure 13: XACML Flow

3.2.2.3 Open Policy Agent (OPA)

OPA is an open-source, general-purpose policy engine. It can unify policy enforcement for APIs. With OPA, it is possible to specify policy as code and simple APIs to offload policy decision-making (from software). This can be used not only in API gateways but also in CI/CD pipelines, microservices, and Kubernetes setups.

In OPA, the policy decisions are generated by evaluating the query input and against policies and data. These policy decisions will help to determine which users can access which resources, which times of day the system can be accessed, etc.

More information is included on the official website for OPA.

3.2.2.4 Speedle+

Speedle is an open-source project to address access control requirements for various applications. It can be used in API management scenarios, cloud-native applications, and legacy applications. This authorization engine can help to externalize access control logic to a policy engine using a fine-grained access control mechanism. Speedle is powered with a policy management service, policy definition language, authorization decision service, and a command-line tool. When securing APIs using a role-based access control (RBAC), Speedle is a good alternative that deserves attention.

This Speedle resources page contains a collection of articles and repositories about it.

3.2.3 Rate Limiting

Allowing unlimited access to APIs is not a good practice (even for legitimate users). If that happens, anyone (who has a subscription to that API) can consume the API, as much as they want at any time. This will become a problem because this will lead to unmanageable loads to the backend service (and the API too). Sudden downtimes might also occur due to the high level of resource consumption.

To avoid these problems, the best solution is to have a rate-limiting mechanism. The limiting conditions can be defined with many conditions such as the number of requests, bandwidth limit, query complexity limit, etc. Either way, the sole intention is to protect the API (and the backend service) while ensuring a fair user experience for all (fair usage policy).

With the right configurations, system administrators can define ideal throttling conditions based on the limits in the backend, limits to the server (API management system), and infrastructure availability.

Enforcing the right rate-limiting to protect APIs will be advantageous in the following scenarios.

- Prevents DDoS attacks - Preventing cybercriminals from flooding a network with so much traffic that it cannot operate or communicate as it normally would.
- Implement API usage plans - This will be beneficial when monetizing APIs and implementing a revenue generation system from APIs.
- Enforce fair usage policies - Makes sure that no one can consume all the allocated resources or bandwidth.
- Prevents systems from over usage - With proper rate-limiting, it is possible to protect APIs and backends from sudden overuse and request spikes.
- Query complexity-based rate limits for GraphQL APIs - When it comes to GraphQL APIs, analyzing the complexity of the query can become a lifesaver to prevent excessive load to the server, database, or network.

3.3 API Protection

This section describes different types of strategies to protect APIs.

3.3.1 CORS Protection

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources to be accessed across domains (outside the domain from which the resource originated).

CORS is used when an API is invoked by web browsers or web apps that are running on a host, which is different from the host of the API. Here, the client application first sends a pre-flight request (options call) to the server. Then, the server will send information such as Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers. These will specify the permitted origins, HTTP methods, and headers.

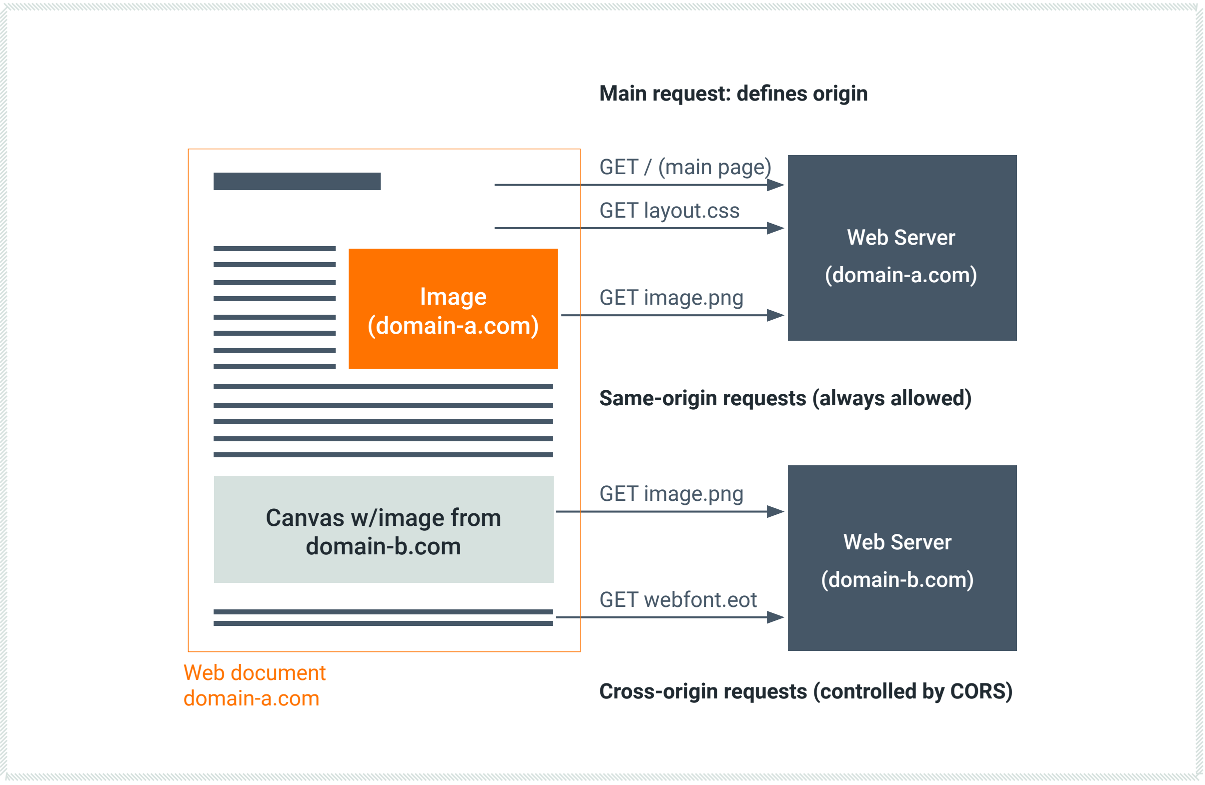


Figure 14: Overview of CORS

The preflight request (options call) is sent before the actual API invocation happens. The same headers should be there in response to the actual request as well. If not, (i.e., a header mismatch) the actual request will not be processed.

3.3.2 Bot Detection

It is important to have advanced security mechanisms to protect APIs from being accessed by bots and hackers. Hackers may try to consume APIs via open ports in the API management system. Usually, this attempt will be made with an open port scan and should be detected at this stage.

These types of attacks should be prevented and information should be sent to the relevant parties to make necessary changes to the system.

Until the actions are implemented, the system should automatically block the originating IP address from accessing any more APIs until further notice. This step will further enhance the security of the system (and other APIs).

3.3.3 Protection against Malicious Payloads

Security attacks are getting more and more sophisticated every day. Hence, detecting and preventing those becomes vital. These days, it is very hard to trust every request that comes to a particular API, even though it comes from a legitimate user. Modern security protocols need to detect and respond to dynamic attacks and the unique vulnerabilities of each API.

In this kind of scenario, enforcing the right security against malicious payloads becomes a must. This way, the system can be safeguarded against intentional or unintentional attacks. For example, some legitimate users might send a payload that can harm the backend service unintentionally. If this request is not prevented, the service will surely experience problems.

When enforcing security mechanisms to protect against malicious payloads, it is necessary to address all kinds of payload-based scanning. This includes protection against JSON message threats, XML message threats, JSON schema-based threats, regular expression-based threats, SQL injections, and script-based (Javascript) attacks. That is the only way to properly ensure the security of the system.

This security mechanism will be able to validate the payload against a predefined rule set and ensure that the backend is safe from harmful requests. With this mechanism in place, API providers and developers can take necessary prevention actions in advance.

A Web Application Firewall (WAF) is another way to ensure protection against attacks caused by malicious payloads. A WAF can protect APIs from a variety of application-layer attacks such as cross-site scripting (XSS), SQL injection, and cookie poisoning, etc.

3.3.4 Data Masking and Data Redaction

When discussing API security in any API ecosystem, the security of user data cannot be ignored. Sometimes, it is necessary to retrieve some user claims for authentication purposes (authorization server). However, that information must be hidden from others, such as the backend service. Data masking becomes a useful solution to address these kinds of scenarios.

When implementing this strategy, there should be a mechanism to filter sensitive information in the API management system. This can be achieved by using a message transformation or message mediation logic. Once that is engaged, sensitive information will be filtered or masked with some other data to hide the exact values. Consumers are ensured that their sensitive information is used only for authentication purposes (and nothing more than that).

Data reduction can be performed in a conditional manner as well. For example, consider a request sent to check if a person is eligible to vote (i.e., people who are over 18 years old can vote). In this case, the API can return the output as a boolean where it indicates the condition - “is this person above 18 years of age”, rather than exposing the actual age of the person. These kinds of modifications can be used to protect sensitive information and mask the data.

3.3.5 Fraud Detection

A security breach in the API ecosystem can happen in many different ways. Sometimes, an attacker can steal a valid credential from a legitimate consumer and try to consume services. If the system solely relies on the credentials (or tokens), the attacker can do anything pretending to be a legitimate consumer. In simple terms, it is hard to fully “trust” each and every request as a legitimate request even though some valid credentials are used.

These types of attacks cannot be simply prevented with token or credential-based authentication protocols. Each API has its own access patterns, which makes it difficult to detect a specific pattern by analyzing large volumes of data manually or by using static policies. AI or unsupervised machine learning algorithms becomes a lifesaver in these critical situations. Moreover, the system should notify the respective people about the attack so they can take extra steps to safeguard their credentials.

With that, APIs can be protected from credential theft and abnormal access patterns. The system needs to be continuously monitored and the algorithm has to be “trained” to detect security threats while making it harder to perform an attack even with a set of valid credentials.

If a token is stolen or a security breach happens (irrespective of all the security mechanisms), then, system administrators should be able to trace the issue to identify which information has been compromised. This is where a logging mechanism comes into play. With the logs, system administrators or engineers should be able to track when the breach occurred, which data was accessed, from which IP it was accessed, etc. This kind of information will be beneficial to improve the security of the APIs in the future. Hence, a comprehensive error logging system becomes a valuable asset to trace down a security breach even after it happens.

3.4 Propagating Security Contexts to APIs

As mentioned in the first chapter, the API gateway is primarily responsible for handling API security.

Most of the time, the user claims and credentials are used at the API gateway level and will not be carried forward after the authentication. However, in some cases, there will be a need to send end-user attributes to the backend for authentication purposes (for some business logic, user information is needed, hence it should be propagated to the backend in a secure manner). A JSON web token (JWT) is the standard way to support these kinds of scenarios. According to the RFC specification, JWT is a JSON object that can be used as a safe way to represent a set of information between two parties. The token is composed of a header, a payload, and a signature. JWT is useful for authentication and information exchange (mostly to forward a set of claims to backend services to further authentication).

When the request reaches the backend service, it can verify the identity of the user who sent the request and grant/deny the request based on the claims.

3.5 OWASP Top 10 API Threats

The advantages that APIs bring can also be a massive liability due to the exposure of major API security vulnerabilities resulting in data theft, unauthorized data manipulation, and downtime to critical business systems, leading to business disruption and monetary loss.

Figure 15: OWASP Security Project

The most prevalent API security vulnerabilities have been identified by the Open Web Application Security Project (OWASP), which is a non-profit foundation that works to improve and establish security standards in the software industry. Refer below for The OWASP API Security Top 10 list, which focuses on the following vulnerabilities. Click the links to read detailed explanations of each security threat and how they can be prevented.

1. Broken Object Level Authorization
2. Broken User Authentication
3. Excessive Data Exposure
4. Lack of Resources & Rate Limiting
5. Broken Function Level Authorization
6. Mass Assignment
7. Security Misconfiguration
8. Injection
9. Improper Assets Management
10. Insufficient Logging & Monitoring

To build a secure API ecosystem, preventing these threats becomes key. At the same time, these standards should be enforced throughout the entire development lifecycle of an API.

3.6 Security Assessments of API Definitions

Accurately identifying potential vulnerabilities in API definitions and addressing them immediately is a growing concern in today's API management arena.

The OpenAPI specification is widely used as the definition language for APIs. By analyzing this definition, it is possible to evaluate the security aspects of a given API. Sometimes, it is important to evaluate this API definition and check if it contains the necessary elements in it.

If this aspect is missed, then there is a possibility to expose a security vulnerability when the API is published. This should be prevented in the early stages of the API development lifecycle.

In the OpenAPI specification, "securityDefinitions" element defines how API clients must authenticate to use API operations. If this section is not defined in the API definition, it means that anyone can use API operations as long as they know the URLs of the operations and how to invoke them. Hence, it is important to identify and prevent this issue before the API is moved to the production environment. Developers can improve the security of the API as soon as they identify the issues in the definition.

3.7 Wrap-up

So far the discussion was about why API security is needed and what are the key facts that need to be considered when securing APIs in an organization.

The content in this ebook has been organized to provide a clear picture of what API security is about and the positive impact it has on the business world.

Accordingly, it is important to thoroughly evaluate the capabilities of an API management solution before deploying it in a production environment. The recommended practice is to choose a product that is well established in the market. Since it's battle-tested, it tends to cover most of the common security threats applicable in that domain. Moreover, even if someone isn't fully aware of different threats, the product guides to create and manage APIs that are secure by design (in most cases).

Conclusion

In most organizations, APIs are being used to achieve important objectives such as:

- To expose internal services within the organization
- To communicate with partners and collaborators
- To expose services to consumers

Depending on the size of the organization, there will be a mix of these API types.

Irrespective of the objective (and the organization type), API security has become a crucial part of the API ecosystem. If this factor is ignored, there can be significant consequences, which could potentially have damaging results on both the business and end-users. Therefore, it is critical to have robust and reliable ways to protect APIs from attacks.

This book explains the factors that need to be considered when developing and evaluating an API management solution that is capable of offering maximum security for the entire API ecosystem.

wso2.com