SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A practical approach to enhance web APIs security using a stateless, open-source, pluggable API gateway

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Federico Dei Cas**

Student ID: 952926
Advisor: Prof. Stefano Zanero
Co-advisors: Mario Petrella (Certimeter) Gianmario Colli (Liquid Reply)
Andrea Moscato (Liquid Reply)
Academic Year: 2022-23

Abstract

With the growing need for enterprises to expose data, services and functionalities and the rise of microservices architectures in modern software development, APIs adoption has increased, becoming a frequent attack vector in security breaches. In this context the role of API gateways has gained importance in the process of securing APIs. API gateways are an API management tool that act as an entry point to manage all traffic and decouple clients from the target services they expose, offering different kind of functionalities that go from simply routing requests to providing foundational security features like authentication and authorization, rate limiting, encryption, request validation and logging, as well as other capabilities that go beyond security aspects. KrakenD is an open-source, stateless API gateway designed to be performant, easily extensible and pluggable with external services. The presented work aims to explore the role of API gateways in the field of API security and the advantages and limitations that characterize KrakenD with its stateless architecture, analyzing the steps that are needed to employ it in common scenarios: routing traffic to an existing API, integrating with an Identity Provider, configuring rate limiting mechanisms, achieving high availability with multiple independent instances and analyzing aspects related to its deployment.

Keywords: API, API Gateway, API security, API management, KrakenD, open-source

Abstract in lingua italiana

Con la sempre maggiore necessità di esporre dati, servizi e funzionalità da parte delle aziende e l'aumento delle architetture a microservizi nello sviluppo software moderno, l'utilizzo delle API è aumentato, diventando un frequente vettore di attacco nelle violazioni di sicurezza. In questo contesto il ruolo degli API gateway ha acquisito importanza nel proteggere le API. Gli API gateway sono uno strumento di gestione delle API che funge da punto di ingresso per gestirne tutto il traffico e disaccoppiare i client dai servizi di destinazione che espongono, offrendo diversi tipi di funzionalità che vanno dal semplice instradamento delle richieste alla fornitura di funzionalità di sicurezza fondamentali come autenticazione e autorizzazione, limitazione del traffico, cifratura, validazione delle richieste e logging, nonché altre funzionalità che vanno oltre gli aspetti legati alla sicurezza. KrakenD è un API gateway stateless open source progettato per essere performante, facilmente estendibile e collegabile a servizi esterni. Il lavoro presentato mira ad esplorare il ruolo degli API gateway nel campo della sicurezza delle API e i vantaggi e i limiti che caratterizzano KrakenD con la sua architettura stateless, analizzando i passaggi necessari per impiegarlo in scenari comuni: instradare il traffico verso un'API esistente, integrarsi con un Identity Provider, configurare meccanismi di limitazione del traffico, ottenere un'elevata disponibilità con più istanze indipendenti e analizzare aspetti relativi al suo deployment.

Parole chiave: API, API Gateway, API management, API security, KrakenD, open-source

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 Background **5**

2.1 API Gateways 5

2.2 KrakenD 6

2.2.1 KrakenD overview 6

2.2.2 The configuration file 7

2.2.3 Extending KrakenD 9

3 Threat Model **13**

3.1 Analysis overview 13

3.2 OWASP API Security Top 10 2023 14

3.3 OWASP API Security Top 10 2019 23

3.4 Evaluation analysis 25

4 Approach **27**

4.1 Approach overview 27

4.2 KrakenD deployment analysis 31

5 Implementation **33**

5.1 Starting from a sample API 33

5.2 Adding the API gateway 33

5.3 Integrating with an IdP 35

5.3.1 Generating JWT access tokens 36

5.3.2 Configuring KrakenD to protect endpoints . . . 36
5.3.3 Testing protected endpoints with Postman . . . 37
5.4 Achieving high availability . . . 38
5.4.1 Distributing ingress traffic with a load balancer . . . 38
5.4.2 Configuring mutual TLS between load balancer and KrakenD . . . 38
5.4.3 Analyzing token revocation in a cluster . . . 39
5.5 Configuring rate limiting . . . 46
5.5.1 Stateless rate limiting on clusters . . . 46
5.5.2 Stateful global rate limiting on clusters . . . 47
5.5.3 Analyzing stateful and stateless rate limits . . . 49
5.6 Enabling monitoring tools . . . 51
5.7 Analyzing the deployment phase . . . 53
5.7.1 Injecting secrets into the configuration file . . . 53
5.7.2 Implementing a CI/CD pipeline . . . 54
5.7.3 Configuring a secure network architecture . . . 55
5.7.4 Benchmarking the gateway overhead . . . 56
5.7.5 Load testing the API gateway . . . 58

6 Conclusions **61**

Bibliography **63**

A Appendix **67**

List of Figures **85**

Listings **87**

List of Tables **89**

Glossary **91**

1 | Introduction

Enterprises today rely on APIs to expose their data and functionalities to business partners, customers, or between different teams inside the company. APIs are a set of definitions and protocols which provide an abstraction to access data and functionalities across different software: [11] in our context we refer to web APIs, a subset of those that are used over a network to access remote resources [51]. Web APIs are typically consumed by IoT devices, web and mobile applications to get access to valuable information; exposing access to this data poses security risks. According to The State of API Security in Q1 2023 by Salt Security *"95% of companies had an API security incident in the last 12 months, API attack traffic grew by 681% while overall API traffic grew 321%."*[48] Since the majority of data breaches are financially motivated [50] and APIs provide a direct access to data that are usually sensitive, common attacks aim at gaining unauthorized access to those data or at disrupting their availability by purposely overloading the exposed services to make them unavailable to legitimate users. The field of API security provides techniques to protect APIs against these attacks. [4] [7]

API gateways are an API management tool that can be employed to enhance the security of APIs, since they act as a single entry point for all traffic, routing requests from the clients to the target services they consume and offering a variety of different functionalities that, from a system design point of view, makes sense to take place at this stage. Security controls are one of these: API gateways usually provide foundational security features like authentication, authorization, encryption, request validation, rate limiting, logging and monitoring. These features are not their sole purpose, as they also provide other functionalities that go beyond security aspects: protocol translation, request aggregation and manipulation, caching, load balancing, and billing. [10] [16] The actual features depend on the specific implementation.

The first generation of API gateways appeared as an abstraction layer to route requests when monolith applications evolved into independent services. API gateways implemented the routing functionality that was originally in the monolith. From a system design point of view, it was useful to implement other centralized features like authentication and

traffic control at this stage, to avoid duplicate functionalities in each service. These advantages have been amplified by microservices architecture and API gateways have evolved, becoming a standard architectural pattern among developers. [34]

In this context, different gateways have emerged. KrakenD is an open-source API gateway written in Go characterized by a stateless architecture and an extensible design. A stateless architecture means that there is no centralized database used to store configuration and everything is managed through a file that is loaded in memory at startup [31]: if we want to make changes in the configuration we have to restart the gateway, but this design comes with performance advantages that can allow to easily scale horizontally with multiple independent instances and leverage the adoption of an immutable infrastructure that makes deployments simpler and safer. [9] [46]

KrakenD community is still relatively small with respect to other open-source API gateways and it's not easy to find valuable information about scenarios in which its architectural patterns are best suited, apart from its official documentation. The aim of this work was to analyze the role of API gateways in enhancing API security, exploring how we can do this with KrakenD: we analyzed the steps that are needed in order to configure it for common scenarios and the advantages and limitations that comes with its stateless architecture, realizing a PoC implementation that could be used as starting point for its employment.

In this chapter we described the context by introducing the field of API security and the concept of API gateways, presenting the motivations behind our work. The rest of the thesis is organized as follows:

- **Chapter 2**. In this chapter we present the main functionalities that API gateways provide and then we focus on introducing the main characteristics of KrakenD.
- **Chapter 3**. In this chapter we introduce the most critical security risks related to APIs from OWASP API security Top 10, analyzing the role of API gateways in mitigating those risks, using KrakenD as example.
- **Chapter 4**. In this chapter we present our approach: starting from the scenario in which we want to place an API gateway in front of an existing sample API, we analyzed how to configure the gateway to route the traffic to the target API and provide foundational security features like integrating with an external IdP for authentication, configuring the gateway to validate the access tokens generated by the IdP, configuring rate limiting mechanisms and achieving high availability with a cluster of multiple instances. Then, we analyzed aspects regarding the deployment

phase: how to automate it with a CI/CD pipeline, how to inject credentials into the configuration file and how to execute load test on the gateway to dimension it correctly according to the traffic we expect to receive.

- **Chapter 5**. In this chapter we present details on how we implemented the approach presented in chapter 4.
- **Chapter 6**. In this chapter we conclude by synthesizing the advantages and limitations that we found about KrakenD and future works that can be done.

2 | Background

In this chapter we analyze the features that API gateways provide and we introduce the main characteristics of KrakenD.

2.1. API Gateways

API gateways are an API management tool that sits between the clients and a collection of backend services, acting as a reverse proxy to route requests and providing different features. Their main ones are [10] [12]:

- **Routing**

  The base functionality of an API gateway is to act as a reverse proxy to forward requests from clients to the target backend services, decoupling them and providing a uniform interface to the consumers.

- **Authentication and Authorization**

  API gateways can handle authentication and authorization to ensure that only the right requests reach the target services. Authorization can be enforced by the API gateway but should be further handled inside the application. [36] [44]

- **Encryption**

  API gateways encrypt traffic with TLS.

- **Traffic management**

  Being the entry point for API traffic, one common feature of API gateways is to put limits in the maximum amount of calls that each user can do in a defined time frame to avoid overloading the system and reduce the probability of Denial of Service attacks in succeding. We can also use the gateway to filter requests in some ways for example filtering by their IP addresses or the user agent.

- **Request and response validation**

API gateways can validate request before they reach the target services and responses before they are sent back to the clients.

- **Data manipulation**

  API gateways can modify requests and responses before they reach their destination: they can filter or mask data and also translate requests and responses between different protocols or aggregate multiple responses from the target services into a single response to be sent to the client requesting it.

- **Caching and load balancing**

  API Gateways can reduce the load on the target services by caching responses and load balancing requests across different backends.

- **Logging and monitoring**

  Since all API traffic goes through the API gateway, it can be used to log its activities and the requests it process, possibly enabling advanced monitoring alerts or integrating with advanced security monitoring tools.

- **Billing**

  If the API are monetized then the gateway can be used to track each user usage and to connect to a billing system to charge them.

2.2. KrakenD

2.2.1. KrakenD overview

KrakenD is an open-source API gateway written in Go that is focused on extensibility and high performance. It is characterized by a stateless architecture that makes it fast and easily scalable, as we can deploy multiple independent instances that don't need synchronization: each of them will have its own configuration file under which we can set everything it needs to work. [16] The gateway is composed by multiple components that are built on top of its core engine: Lura[1], an open framework to assemble API gateways that was developed by KrakenD and later donated to the Linux foundation. Other than the open-source version KrakenD also offers an enterprise paid version that has additional proprietary functionalities. [16]

[1] https://github.com/luraproject/lura

```
1 {
2     "$schema": "https://www.krakend.io/schema/v3.json",
3     "version": 3,
4     "endpoints": [],
5     "extra_config": {}
6 }
```

Listing 2.1: KrakenD's configuration file main structure, extracted from [14]

2.2.2. The configuration file

Since everything in KrakenD is managed through the configuration file, in this section we provide an overview of its main structure. The configuration file is parsed when the gateway is started and is then loaded in memory. KrakenD supports different file formats, the default one is JSON. The main structure of the configuration file is composed by: [14]:

- **$schema.** Defines the JSON schema[2] to validate the configuration automatically. This part is optional. [14]
- **version.** Defines the version of the configuration file. [14]
- **endpoints.** Defines an array of objects where each object represents an endpoint that the gateway exposes to the clients. [14]
- **extra_config.** Defines additional configurations that are not managed by the core KrakenD engine. [14]

The JSON structure of the configuration file is shown in Listing 2.1. The first advantage in this approach is that the configuration file (and so our API definition) can be versioned. With the Go templating system[3] we can separate the configuration file into different sub-files, and we can inject environment variables or files inside the configuration to load secrets or environment specific settings sharing the same base configuration file. [19] In chapter 5 we provide one possible approach to inject credentials into the configuration file from secrets management tools.

The endpoint structure

The `"endpoints"` part in the configuration file specifies a list of endpoints the gateway exposes. Each object of the list represents a single endpoint and for each of them we need to specify one or more backends, since the gateway can act as an aggregator to merge

[2] `https://github.com/krakendio/krakend-schema`
[3] `https://pkg.go.dev/text/template`

```
"endpoints": [
    {
      "endpoint": "/users/{user}",
      "method": "GET",
      "backend": [
        {
          "url_pattern": "/users/{user}",
          "method": "GET",
          "host": [
            "https://jsonplaceholder.typicode.com/"
          ]
        }
      ]
    }
  ]
```

Listing 2.2: KrakenD's endpoints configuration sample

responses from multiple backends. Inside each backend we then need to specify one or multiple host, if we want the gateway to load balance the requests across those. [14]

The sample configuration shown in Listing 2.2 is a basic example that makes the gateway expose an endpoint `/users/{user}`, where `{user}` is a parameter used to retrieve information about the user specified in the path, returning the response from the GET request to the target backend from the `"host"` array. [14]

The extra_config structure

The `extra_config` is used to load the configuration of components that are not part of the core engine, and can be placed at different levels depending on their scope; each component can support different scopes or can be built specifically for a single scope. Under its namespace, which identifies the component uniquely, we can set specific additional configurations, as shown in Listing 2.3. The possible levels are: [14]

- **service.** This level is used to set configuration for components that can either influence the gateway globally or on every request. For instance, we can configure the gateway to export metrics to an external database or set a global rate limit for every request. [14]
- **endpoint.** This level is used to set configuration for components that target a specific endpoints that KrakenD exposes. In this scope we can configure, for instance, a rate limit set for a specific endpoint. [14]

```
{
    "extra_config": {
      "component-1-namespace": {
        "some": "config"
      },
      "component-2-namespace": {
      }
    }
}
```

Listing 2.3: KrakenD's extra_config structure, extracted from [14]

- **backend.** This level is used to set configurations for components that target a single backend that the gateway contacts. In this scope we can set configurations for how the API gateway interacts with the target backend: for instance, we can rate limit the number of calls that KrakenD can make to a specific service. [14]

2.2.3. Extending KrakenD

There are multiple ways to extend KrakenD with additional features. The first naive approach would be to make sure that what we are trying to achieve is not already possible using an already existing component or a combination of the available ones. If our use case scenario is not covered, then we can implement our additional logic in two ways: [30]:

1. By creating our own custom component or editing an existing one, forking the original repository[4]
2. By injecting a custom plugin[5] or a LUA script[6] that allows to integrate functionalities into different parts of the processing

Since KrakenD is a collection of multiple components merged together with the core engine[7], where each component is shipped in a separated repository, one possible approach is to fork the original main repository[8], create our own custom component and add it to our fork. However, the recommended approach is to use plugins because in this way we don't need to fork the original repository, as they are side loaded along with the official binaries. Plugins are soft linked Go libraries that can be injected into different parts of the request and response processing. [30] According to the official documentation, there

[4] https://github.com/krakendio/krakend-ce
[5] https://pkg.go.dev/plugin
[6] https://www.lua.org/about.html
[7] https://github.com/luraproject/lura
[8] https://github.com/krakend/krakend-ce

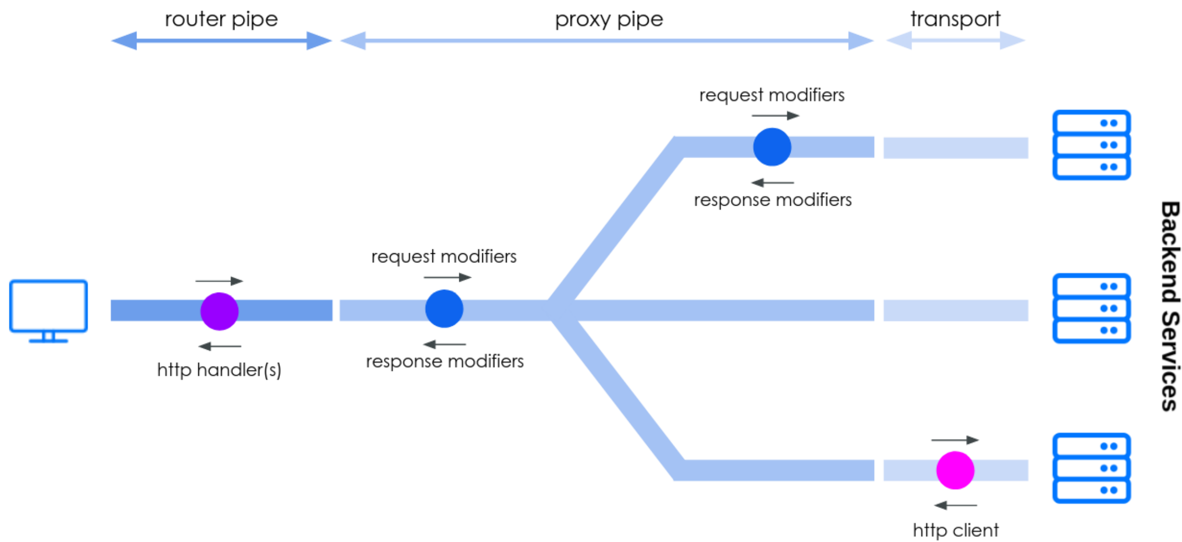


Figure 2.1: KrakenD's plugins types, Extracted from [30]

are 3 types of them:

1. **HTTP server (or HTTP handlers).**[9] These plugins inject functionalities as soon as the request hits the gateway or just before the response is sent back to the client. With these we can add functionalities that influence the gateway globally. From the perspective of KrakenD, they are black boxes that expose the request and response through an interface. We can chain multiple HTTP server plugins together. [30]

2. **HTTP client.**[10] These plugins inject functionalities between KrakenD and the target backend. We can modify the request just before it reaches the service or as soon as the response is received back by the gateway. Differently from the HTTP server plugins, we cannot chain together multiple plugins of this type. [30]

3. **Request/Response modifier.**[11] These plugins allow to intercept and modify the request or the response at endpoint or backend level. [30] These two levels are different because KrakenD does not necessarily create a 1:1 correlation between an endpoint exposed by the gateway and a target backend, for instance because in a microservice based application the request could be composed by aggregating the response from multiple backends, returning a single response to the client.

The different types of plugins and their scopes are schematized in Figure 2.1.

[9] https://www.krakend.io/docs/extending/http-server-plugins/
[10] https://www.krakend.io/docs/extending/http-client-plugins/
[11] https://www.krakend.io/docs/extending/plugin-modifiers/

Another possible way to extend the gateway with additional logic is to use LUA scripts[12] that, like plugins, can be placed in different parts of the request and response processing. [16] However, LUA scripts are much slower with respect to compiling custom plugins and side loading them along with KrakenD: according to the developers a Go plugin is at least 10 times faster than a LUA script [30], therefore in our work we did not use those since we can overcome the limitations of having to recompile plugins manually at each change with a CI/CD pipeline.

[12] https://github.com/krakend/krakend-lua

3 | Threat Model

In the current chapter we analyze the role of API gateways in the field of API security by identifying the most critical security risks related to APIs and the role that API gateways can pose in mitigating those. Risk is the statistical evaluation of the economic damage posed by the presence of vulnerabilities, which could be leveraged to perform an exploit. [45]

3.1. Analysis overview

In our analysis we assume to place an API gateway in front of a vulnerable API, with authentication managed by the gateway together with an Identity Provider that conforms to the OIDC standard and communication from clients to the API gateway that happens through a secure channel. We assume the user has authenticated with the Identity Provider and then its client, on behalf of the user, authorized with specific scopes that defines which resources it has been provided access to, receiving a JWT access token that the client subsequently presents to the API gateway. The target API can only be accessed through the gateway. The requests and responses are in JSON format.

The analysis is schematized in Figure 3.1 and synthesized in the list below:

1. The client authenticates to the IdP and requests authorization for specific scopes on the resources of the target API.
2. The client receive a JWT access token that is signed by the IdP and contains the scopes which it has been provided access to.
3. The client present the token to the API Gateway attaching it in the request for a resource.
4. The API gateway verify the token and, if valid, forwards the request to the API.
5. The API reply to the API gateway with the response.
6. The API gateway forwards the response to the client.

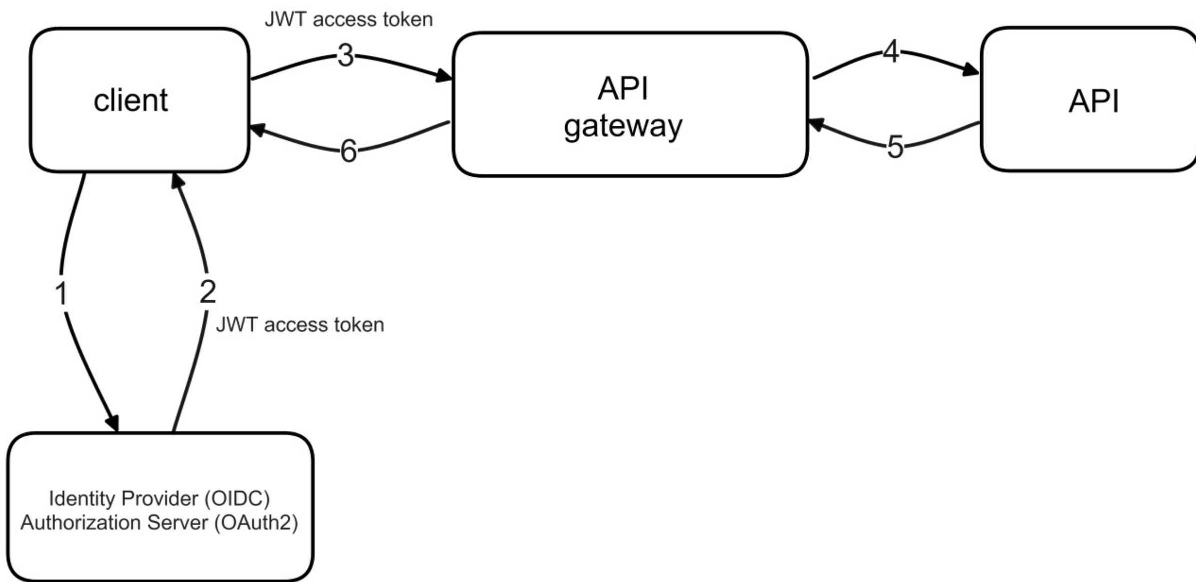


Figure 3.1: Scenario of our analysis

OWASP is a community driven non-profit organization dedicated to software security; one of their projects is OWASP API security Top 10, which is a report that lists the most critical security risks related to APIs along with the strategies that can be used to mitigate them. [41] Even though the presented vulnerabilities are more related to the design of APIs themselves and an API gateway alone is not the sole strategy to protect against them, since our analysis is focused specifically on the role of API gateways in the field of API security we assume that we can only act on the gateway, analyzing how its role is relevant in mitigating those risks, using KrakenD as an example.

At the end of the chapter we provide a qualitative scale to define the relevance of API gateways in the mitigation of each presented risk.

3.2. OWASP API Security Top 10 2023

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. [43]*

These vulnerabilities refers to the lack of or misimplementation of proper authorization checks at object level. An example scenario could be an API that exposes information

about users, in which each user can access their own personal data requesting the endpoint GET `/user/{user}` where `{user}` is a parameter that represents its unique identifier. If the API only validates that the user is authenticated without further authorization checks, an attacker could guess other IDs and get information of users which he should not have access to.

The API gateway can enforce some authorization checks as a first line of defense; the authorization should be then validated into the application itself. Delegating all authorization logic to the gateway would not be ideal from a system design point of view. [36] Also, having the authorization server in the identity provider handle all authorization logic using the gateway to validate it would still not be ideal. [3] In our scenario the authorization server would give authorization to read users information, but the specific object level authorization of which users information the token owner can access should be handled in the application logic and possibly enforced by the gateway: we could use the API gateway to check that the endpoint can only be accessed by the user who has the same ID of the endpoint he is requesting, by inspecting the JWT token to get his ID and confronting it with the ID included in the request. However, according to the latest update by OWASP, this approach only covers a small subset of cases in which a BOLA vulnerability can be leveraged. [43]

In KrakenD, using the `validation/cel`[1] component we can access the JWT payload to implement the logic needed to check that the user requesting access corresponds to the user specified in the path, allowing us to discard the request if not valid. Assuming that the `sub` claim of the JWT inside the payload contains the ID of the user, we can write a CEL[2] expression to validate that this ID correspond to the user parameter in the request. Since the CEL validator component can't evaluate the JWT and the request parameters at the same time[3], as shown in Figure 3.2, one possible approach is to attach the user ID extracted from the JWT payload as an HTTP header using the `"propagate_claims"` option, and then check that that header correspond to the `{user}` parameter that is received in the request. A configuration sample of this approach is shown in listing 3.1. Note that:

- In this approach we also have to forward the X-User input header, as KrakenD do not forward any headers explicitly[4].
- We can also use a custom claim inside the JWT token payload to validate the `{user}`

[1]https://github.com/krakendio/krakend-cel
[2]https://github.com/google/cel-spec
[3]https://github.com/krakendio/krakend-ce/issues/335
[4]https://www.krakend.io/docs/endpoints/parameter-forwarding/

```
{
    "endpoint": "/users/{user}",
    "method": "GET",
    "input_headers": [
        "X-User"
    ],
    "extra_config": {
        "auth/validator": {
            "alg": "RS256",
            "jwk_url": "https://auth-server/.well-known/jwks.json
               ",
            "propagate_claims": [
                ["sub", "X-User"],
            ]
        },
        "validation/cel": [
            {
                "check_expr": "req_params.User == req_headers['X-
                   User'][0]"
            }
        ]
    },
    "backend": [
    ...
    ]
}
```

Listing 3.1: KrakenD's CEL validation rule example to check that the "sub" claim inside the payload correspond to the {user} parameter of the request

id parameter if the `"sub"` claim does not correspond to the id of the user in our endpoint.

- `req_headers` returns an array [17] because there could be multiple HTTP headers with the same name.

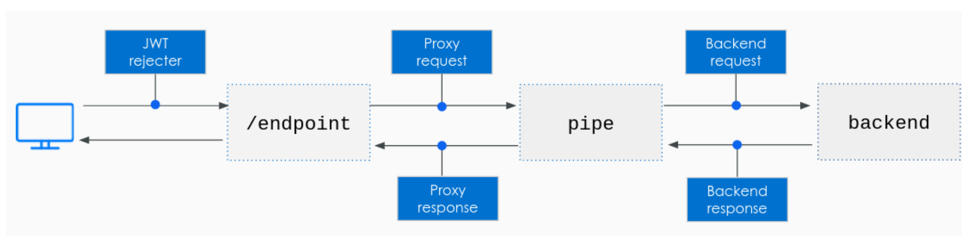


Figure 3.2: Where CEL evaluation can take place, Extracted from [17]

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. [43]*

These types of vulnerabilities are related to the lack of or misimplementation of the authentication mechanisms and could allow an attacker to authenticate as any user, stealing their data or performing sensitive actions on their behalf. [43]

In our analysis the authentication is managed by the Identity Provider together with the API gateway. The role of the gateway in this scenario is to validate that the access token received is authentic and has not been tampered, validating its signature, and checking that it is not expired. The access token provides proof that the user has authenticated with the IdP and specifies inside the payload which authorization has been granted by the authorization server. Failing to validate the token could result in broken authentication: for instance, if the JWT token signature is not verified by the gateway, an attacker could change the `''sub''` claim inside the JWT payload that represents the user ID, authenticating as another user. [1] If the API gateway manages the endpoints to authenticate, then it should also handle appropriate rate limits for those endpoints.

Since KrakenD is stateless it cannot generate the tokens itself but has to rely on an external self hosted or SaaS Identity Provider that conforms to the OIDC standard. KrakenD can then be configured to validate the tokens. [22] In chapter 5 we present how we integrated KrakenD with Auth0, a SaaS identity provider, to generate access tokens and how to configure the gateway to validate those.

API3:2023 Broken Object Property Level Authorization

> *This category combines API3:2019 Excessive Data Exposure[5] and API6:2019 - Mass Assignment[6], 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. [43]*

This risk refers to two different scenarios: the first is when the API returns more data than it should, for instance relying on the client to filter that data. In this case, an

[5] `https://owasp.org/API-Security/editions/2019/en/0xa3-excessive-data-exposure/`
[6] `https://owasp.org/API-Security/editions/2019/en/0xa6-mass-assignment/`

attacker could simply call the API directly to see the excessive data. The second scenario is when the API parameters are tied directly to the underlying object implementation; in this case, an attacker could reach functionalities which he should not have access to. [43]

API gateways can impact the mitigation of these vulnerabilities. For the first scenario we could filter unwanted parameters: if the API is developed in a generic way returning all data and we want to differentiate the amount of data returned based on the audience, we can use the API gateway to expose different endpoints that return different data checking the audience claim inside the JWT access token of the client who is requesting the endpoint. For the second scenario, we can validate the requests to prevent unintended parameters to reach the target API. [36]

In KrakenD, for the first scenario we can expose separated endpoints based on the client requesting it, using the `auth/validator`[7] component to differentiate by the `‘‘audience’’` claim inside the JWT payload, and then use the `allow list`[8] or `deny list`[9] to filter the response received from the target service with a whitelist or blacklist approach, filtering the data that we should not expose to the clients. A configuration sample of this approach is shown in listing 3.2. For the second scenario we can use the `validation/json-schema`[10] component to provide validation rules for the requests body, allowing only the ones that conform the schema to reach the target backend(s).

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. [43]*

This risk refers to the lack of or misimplementation of mechanisms to limit the amount of resources that the API exposes access to. These kinds of vulnerabilities can happen in different ways, from a simple lack of rate limiting, to more tricky scenarios more related to the API logic itself, like the number of allowed operations in a single API call or the maximum upload file size. [43]

With an API gateway we can implement appropriate rate limiting and throttling mechanisms to reduce the number of API calls that each user can make. For long term rate

[7] `https://github.com/krakendio/krakend-jose`
[8] `https://www.krakend.io/docs/backends/data-manipulation/#allow`
[9] `https://www.krakend.io/docs/backends/data-manipulation/#deny`
[10] `https://github.com/krakendio/krakend-jsonschema`

```
1  {
2      "endpoint": "/users/{user}",
3      "method": "GET",
4      "extra_config": {
5          "auth/validator": {
6              "alg": "RS256",
7              "audience": ["AUDIENCE"],
8              "jwk_url": "https://auth0-server/.well-known/jwks.json
                 "
9          },
10     },
11     "backend": [
12         {
13             "url_pattern": "/users/{user}",
14             "host": [
15                 "https://jsonplaceholder.typicode.com"
16             ],
17             "allow": [
18                 "name"
19             ]
20         }
21     ]
22 }
```

Listing 3.2: KrakenD's sample endpoint configuration to filter response using the allow list feature, filtering the response to return only the "name" field

limiting we can enable user quotas to restrict the amount of requests that can be done over a longer span (e.g. a daily/weekly/monthly quota). [36] We can set a maximum size for payloads and set timeouts and restrict resource consumption by using containers or other solutions that allow us to easily limit resource consumption. [43] [6]

In KrakenD, we can configure rate limiting mechanisms for each exposed endpoint using the `qos/ratelimit/router` component[11], either a general rate limit or a rate limit per client which is identified by its IP address or by a configurable HTTP header. Since the gateway is stateless, the counters used to handle rate limits are stored in memory. [24] This has two main consequences: first, we cannot synchronize counters across multiple independent instances of the gateway, second, if the gateway crashes and restarts, its counters will be reset. For user quotas we need to integrate the gateway with some external service because its stateless architecture does not allow it natively. To do that, we can implement a custom plugin. Since KrakenD is stateless, we can easily containerize it with an immutable docker image with the configuration file embedded in it. Using containers we can set maximum resource limits consumption to avoid the system reaching failure. [6] In chapter 5 we present possible approaches that we tested to enable rate limiting mechanisms and containerize KrakenD using an immutable Docker image with the configuration file embedded in it.

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. [43]*

This risk refers to the lack of or misimplementation of role based access control mechanisms.

API gateways can be employed to implement role based access control mechanisms; as already stated in 3.2, the gateway should be used to enforce authorization checks that are then further handled in the application.

In KrakenD, we can validate roles inside the JWT tokens, issued by the Identity Provider: with the `auth/validator`[12] component we can automatically check the roles claim in the JWT payload to ensure that the user requesting access to the specified resource has the necessary authorizations. In this sample configuration, we specify the `roles_key` to

[11] `https://github.com/krakend/krakend-ratelimit`
[12] `https://github.com/krakendio/krakend-jose`

```
{
    "endpoint": "/admin",
    "method": "POST",
    "extra_config": {
        "auth/validator": {
            "alg": "RS256",
            "roles_key": "roles",
            "roles": ["admin"],
            "jwk_url": "https://AUTH0_DOMAIN/.well-known/jwks.json
                "
        }
    },
    ...
}
```

Listing 3.3: KrakenD's JWT role validation endpoint example

define the name of the object to look for `"roles"` claim inside the JWT payload, and only accept those which contain the `"admin"` role. The endpoint configured this way will check if the JWT contains at least one of the roles specified in the `"roles"` key. [22] With the `cel/validator` component, presented in API1:2023, we can also implement a more advanced role based authorization logic, if needed.

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. [43]*

This risk refers to business flows that can be used excessively with automated methods like scripts. [43]

In this scenarios we can leverage API gateways as a first line of defense by implementing rate limiting mechanisms and providing a base protection against automated bots.

In KrakenD, as presented previously in API4:2023, we can implement rate limiting mechanisms. With the `security/bot-detector`[13] component we can provide a first basic level of defense by analyzing the user-agent in the request. However, this mechanism provides just a basic level of protection as it can be easily circumvented by an attacker that could simply spoof the user agent.

[13] `https://github.com/krakendio/krakend-botdetector`

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. [43]*

This risk is related to the absence of validation mechanisms for the user supplied input, that could be used to let the server perform an action on behalf of the user. [43]

To mitigate these flaws we could leverage an API gateway to validate user input. The application code itself should validate every input, regardless of the fact that this is validated or sanitized by other components of the stack.

KrakenD does not provide advanced input validation and sanitization features, but, using the `krakend-jsonschema`[14] validator component we can leverage regex to provide input validation on the request body. To validate query and path parameters, or even headers we can use the `validation/cel`[15] component. In both cases we can also use a plugin to implement a customized validation and sanitization mechanisms.

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. [43]*

This risk can be present in many parts of the stack and refers to a different variety of misconfiguration scenarios that can happen either with or without an API gateway.

Since all APIs are exposed through the API gateway, we can reduce the possibilities of security misconfigurations by having a single entity to expose and reducing their chances by running automated tests.

In KrakenD, we can use the `check`[16] and `audit`[17] tools to respectively look for syntax errors or security misconfigurations in the configuration file. These tools can be integrated in a CI/CD pipeline to reduce the chances of human error.

[14] https://github.com/krakendio/krakend-jsonschema
[15] https://github.com/krakendio/krakend-cel
[16] https://www.krakend.io/docs/configuration/check/
[17] https://www.krakend.io/docs/configuration/audit/

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.[43]*

This risk is common when for backward compatibility old versions of APIs are left accessible. In this scenario they provide an easy path for attackers. [43]

API Gateways are responsible for exposing the endpoints to the clients, and can help in mitigating this risk by providing ways to ease the management of APIs. However, the mitigations are related to how APIs are managed, and API gateways play part in this process as a tool but ultimately this risk is related to how these tools are used.

In KrakenD, since all the exposed endpoints are specified in the configuration file, we can version it to allow for a more structured API management. We can also implement a CI/CD pipeline to update automatically the changes in the configuration file by redeploying the gateway. In chapter 5 we provide a possible approach to implement a CI/CD pipeline to leverage these advantages.

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. [43]*

This risk refers to the lack of input validation mechanisms when relying on data from third-party services. [43]

The role of API gateways is limited in mitigating this risk, since these vulnerabilities rely on a compromised third-party service integrated with our API, and this path may be outside the API gateway scope. If the third-party API traffic pass through the gateway, then we can implement input validation mechanism in the API gateway, as presented in API7:2023.

3.3. OWASP API Security Top 10 2019

These risks were present in the previous version of OWASP API Top 10 and are still worth mentioning, as not being part of the new OWASP version does not imply that these risks

are less critical but rather less common because organizations adopted mitigations that led those to slip out of the top 10 in the newer version.

API8:2019 Injection

> *Injection flaws, such as SQL, NoSQL, Command Injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization. [42]*

This risk refers to a variety of scenario in which the user supplied input is not validated or sanitized and causes the execution of unallowed commands. [42]

The way API gateways can mitigate this risk are the same as SSRF. Generally speaking, the user input that can reach the target API can be inside query strings, path parameters, request body or HTTP headers. We can use the API gateway to validate and sanitize all this input.

In KrakenD, we have multiple ways to implement our own logic to validate user input, as presented in API7:2023. To access query strings, parameters and HTTP request headers and validate those we can use the `cel/validator`[18] component. To access the request body we can use the `validation/json-schema`[19] component to validate the json schema of the request. In both of these cases, we can also implement a custom plugin and access the request directly and implement our own custom solution.

API10:2019 Insufficient logging and monitoring

> *Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems to tamper with, extract, or destroy data. ... [42]*

This risk refers to the lack of or not effective implementation of logging and monitoring features that could allow attacks to go unnoticed. [42]

API gateways are crucial in mitigating this risk since they route all incoming API traffic therefore logging and monitoring is one of their most important features.

Since KrakenD is stateless, it must integrate with external services to export logs[20], met-

[18] `https://github.com/krakendio/krakend-cel`
[19] `https://github.com/krakendio/krakend-jsonschema`
[20] `https://github.com/krakendio/krakend-gelf`

Value	Title	Description
0/3	None.	This risk can not be mitigated at all with an API gateway.
1/3	Minimal.	The role of API gateways in mitigating this risk is minimal.
2/3	Partial.	The API gateway can provide a partial contribution to mitigate this risk.
3/3	Extensive.	An API gateway pose a significant role in mitigating this risk.

Table 3.1: Qualitative scale definition

rics[21] and traces[22]. In chapter 5, we provide a possible approach to integrate KrakenD with external services to achieve this goal.

3.4. Evaluation analysis

In this section we provide an evaluation on the role of API gateways in mitigating the presented risks, using a qualitative scale. The scale that we used is shown in Table 3.1.

In Table 3.2 we provide a rating on the effectiveness of API gateways in mitigating each risk. Our finding is that an API gateway can mitigate some of the most critical risks, providing a first line of defense and managing important security features like authentication, rate limiting, input validation and monitoring. However, the majority of the presented risks are related to the design and implementation of APIs and an API gateway alone is not enough to mitigate these risks, even though it can provide an effective extra layer of security, contributing to the overall API security posture.

[21]https://github.com/krakendio/krakend-metrics
[22]https://github.com/krakendio/krakend-opencensus

Risk	Value	Motivation
API1:2023	2/3	The role of API gateways in mitigating this risk is useful but limited, as the object level authorization checks should be handled inside the application logic. However the gateway is still responsible to validate the access token to ensure that it's authentic and not been tampered and possibly enforce the authorization checks, contributing to the overall mitigation of this risk.
API2:2023	3/3	In our scenario, the API gateway and the Identity Provider are the main responsible for authentication. Specifically, the role of the API gateway is to validate that the access tokens are authentic and have not been tampered, ensuring the client accessing the API has been successfully authenticated. Even though the API gateway may not handle the authentication process directly, its contribution is crucial to ensuring that only authenticated requests can reach the target API.
API3:2023	2/3	The decoupling that API gateways provide can mitigate this risk, either by acting on the output or on the input received from the client. However, to fully mitigate this risk the vulnerabilities should be handled in the application itself.
API4:2023	3/3	API gateways are the element in the system architecture that is directly exposed to the client and handle all incoming API traffic therefore their role in mitigating this risk is crucial.
API5:2023	2/3	API gateways can enforce RBAC checks. However, the validation should be further handled in the application itself.
API6:2023	1/3	API gateways can be used to mitigate some of these vulnerabilities by enforcing access control policies and reducing the possibilities of automation with basic protection features provided by rate limiting, IP filtering and bot detection. However, they provide only basic capabilities to handle this risk.
API7:2023	2/3	The API gateway can be used to enforce input and output validation mechanisms to reduce the chances of abuses and reduce the attack surface by providing a unified interface instead of exposing each service directly to the client. However, the mitigation that they provide is still limited.
API8:2023	1/3	API gateways can reduce this risk by providing a single entity to expose and therefore reducing the chances of security misconfigurations. However security misconfigurations can still happen independently of using or not an API gateway and the mitigation could be to use automated tools to check for misconfigurations.
API9:2023	3/3	API gateways are a crucial tool in mitigating this risk since they handle all the exposed endpoints, even though they are just a tool and therefore this risk still depends on how the API gateways are used to mitigate this risk, meaning how the organization handles the overall API management lifecycle.
API10:2023	1/3	The consumption of third-party APIs can create a path outside the scope of the gateway therefore this risk should be handled in the application itself. If the path goes through the API gateway then we can use input validation mechanisms to help mitigate this risk.
API8:2019	2/3	API gateways provide input and output validation features as a first line of defense against injections. However these vulnerabilities should be also handled in the application.
API10:2019	3/3	API gateways manage all API traffic, therefore they are the best place to handle centralized logging and monitoring features.

Table 3.2: Evaluation analysis of the role of API gateways in mitigating most critical API security risks

4 | Approach

In this chapter we present the approach that we used to test the main capabilities of KrakenD, with the different scenarios that we analyzed. We started from a sample API and then analyzed how to protect it testing the most important features that KrakenD provide, eventually extending it with additional functionalities if needed. We tested the functionalities locally, deploying KrakenD on a local Kubernetes cluster using Docker and Minikube, because in a real scenario we would deploy a Docker image on Kubernetes. For the external services to integrate we used Docker Compose, because in a real scenario the external services may or may not be under our control, meaning outside the Kubernetes cluster.

4.1. Approach overview

We started from a sample API protected with a basic auth authentication scheme. The initial step is schematized in figure 4.1. The sample API is used for mocking a real API: we chose the basic auth authentication because a frequent use case in enterprises is to use the API gateway to connect to legacy APIs that use this authentication scheme.

Routing

We placed the API gateway in front of the API, configuring it to expose the same endpoints that the API provides and configuring the gateway to authenticate with the target API automatically. This scenario is schematized in figure 4.2. From now on we call the API through the API gateway and the target API does not have to be exposed directly as it can live behind KrakenD.

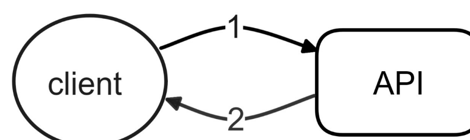


Figure 4.1: Initial step

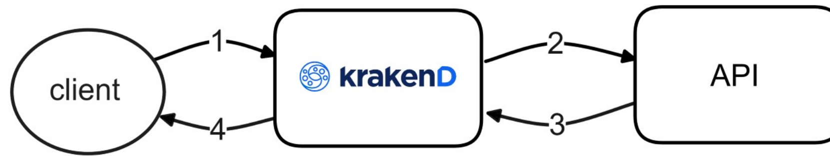


Figure 4.2: Step 1

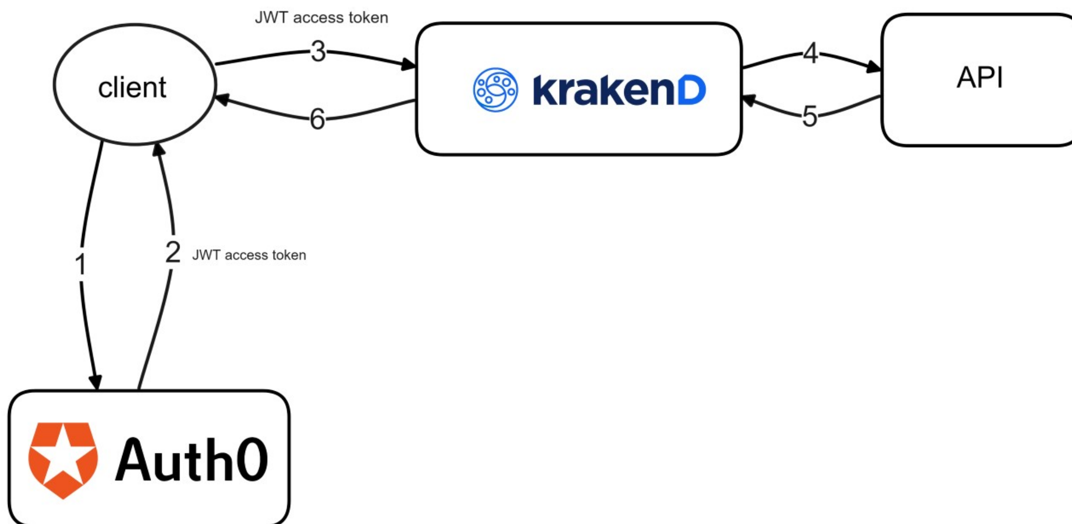


Figure 4.3: Step 2

Authentication and authorization

We examined how to enable authentication and authorization mechanisms using an external IdP to generate JWT access tokens, and configured the API gateway to protect the endpoints by verifying the tokens. The step is schematized in Figure 4.3.

High availability

We tested how to deploy multiple independent instances of KrakenD where ingress traffic is distributed by a load balancer, to achieve high availability. The load balancer availability is out of our scope. In this scenario the API gateways can only be called through the load balancer. This step is shown in Figure 4.4.

Rate limiting

We analyzed how to configure the API gateway with different rate limiting mechanisms in three scenarios, as shown in Figure 4.5.

In the first scenario we tested the rate limiting features for a single instance with the

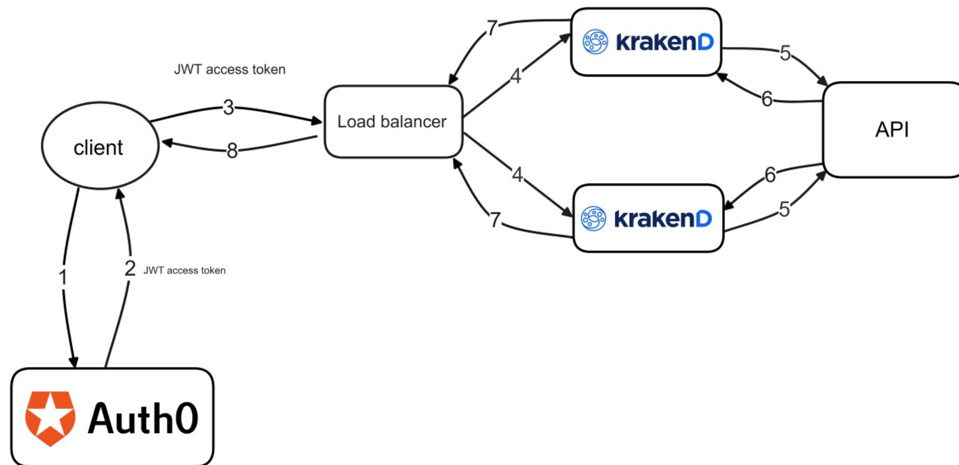


Figure 4.4: Step 3

`qos/ratelimit/router`[1] component through which we can set an absolute limit for an endpoint or a limit for each client on the same endpoint. [24] The clients can be identified by their IP address or by a configurable HTTP header that identifies a user uniquely and the limiting strategy is based on the token bucket algorithm [26].

In the other scenarios we analyzed how to handle rate limiting in case of a cluster of independent instances where ingress traffic is distributed by a load balancer: the problem that arise in this case is that the counters used by the rate limiter are not synchronized. One possible approach, used in the second scenario, is to split the total desired rate limit into different sub-limits for each instance, based on the cluster size and the algorithm used by the load balancer. [32]

In the third scenario we implemented a custom plugin to synchronize counters using a shared redis database to handle rate limiting. KrakenD developers use Redis to offer a global cluster rate limit feature in the enterprise version [28]. Redis is suited for rate limiting because it has builtin commands that allows to handle counter access thread safe and set a time limit after which key will automatically expire [47]. As an in-memory database it is faster than a traditional on-disk database, allowing us to limit the additional overhead requested by this approach. The updated architecture is shown in Figure 4.6.

Observability

We tested how to configure the API gateway to export logs, metrics and traces to external services. These are known as the pillars of observability and help us in having a clear view of the system to continously monitoring it when it's operating. [40] The final architecture

[1]`https://github.com/krakendio/krakend-ratelimit`

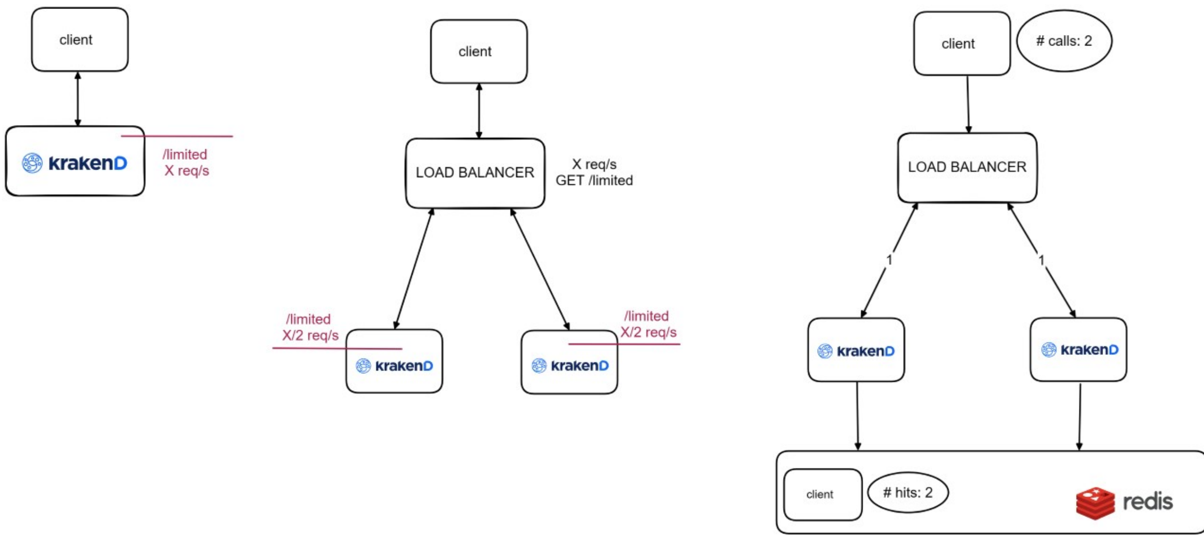


Figure 4.5: Step 4 - Rate limiting scenarios

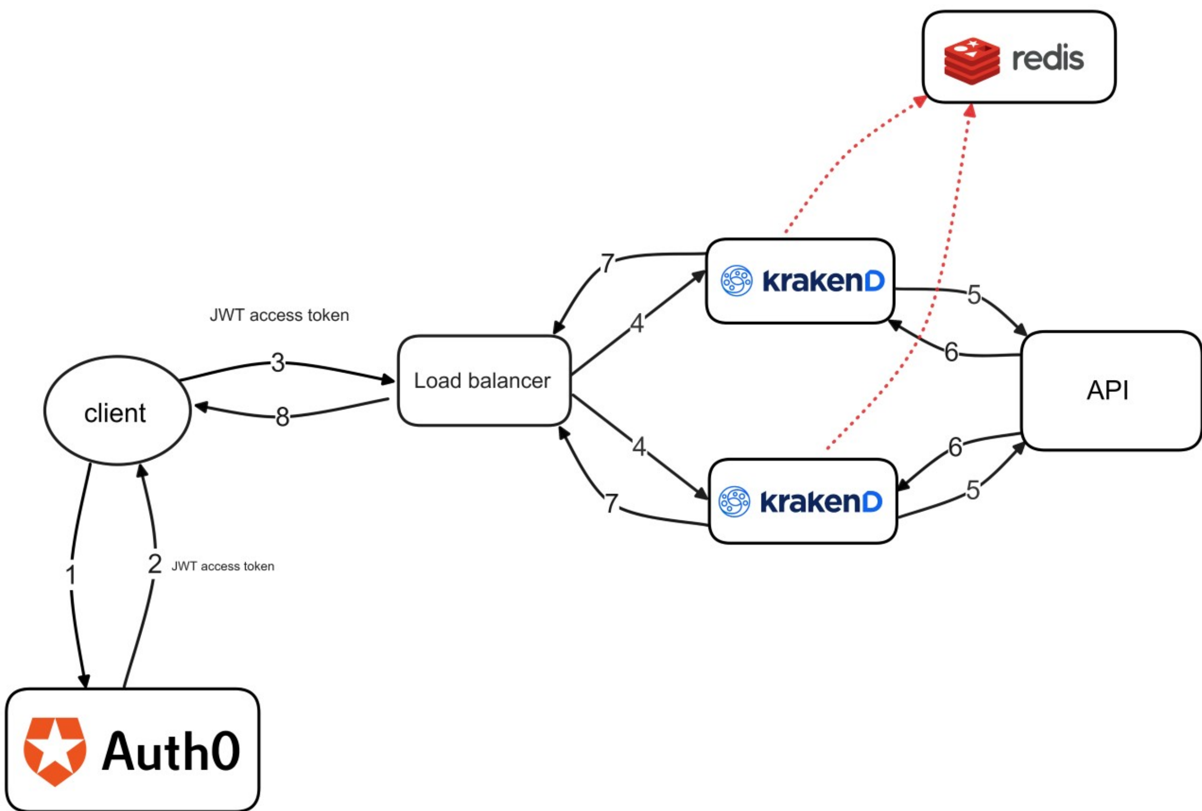


Figure 4.6: Step 4

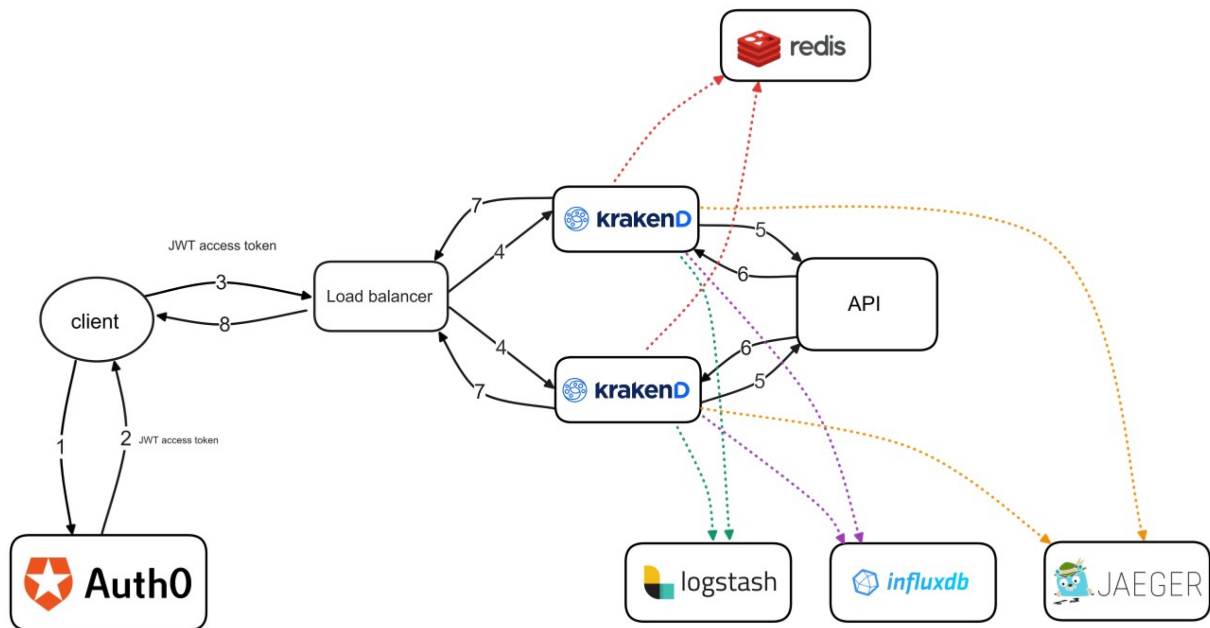


Figure 4.7: Step 5

is shown in Figure 4.7.

4.2. KrakenD deployment analysis

We analyzed aspects regarding the deployment of KrakenD:

We tested how to configure a CI/CD pipeline to deploy the gateway instances automatically. Since KrakenD is stateless, we need to restart it when making changes to the configuration file. In our scenario, we tested to generate an immutable docker image with the configuration file embedded in it, therefore we must redeploy the gateway to apply changes to the configuration. In this scenario, having a CI/CD pipeline to handle the deployment is beneficial, as we can version the configuration file and possibly configure the pipeline to automatically redeploy the gateway when a change is made, executing tests to reduce the risks of human error and security misconfigurations. In our case both Jenkins and the gateway were deployed locally therefore we did not enable automatic redeployments.

We saw a possible approach to inject credentials into the configuration as files that are injected into the pod by Kubernetes secrets. Since the credentials should not be put in the version control systems, we analyzed a possible way to inject secrets using secret management tools and then configure KrakenD to dynamically load them from files.

We analyzed approaches on how to load test the API gateway. First, we tested the

additional overhead of the different functionalities by load testing only the API gateway health endpoint directly, adding one functionality at a time. Then, we tested an approach to add more structured load tests that would allow us to simulate users authenticating to the IdP and calling different endpoints on the API gateway. Since in a realistic scenario the API gateway has to handle traffic for different microservices or even multiple APIs, this phase is an important aspect of the deployment, as introducing an API gateway should not mean introducing a new bottleneck in the architecture.

5 | Implementation

In this chapter we present how we implemented the solution using the approach that we presented in chapter 4.

5.1. Starting from a sample API

First, we developed a sample REST API[1] that would act as our base case. The API expose basic CRUD operations on a list of users, each user contain an `"ID"`, a `"name"` and an `"information"` field and is protected by a basic auth authentication scheme. This API was just for testing purposes to generate mock responses since the focus of our work was on testing the features of the API gateway and not the API itself. We configured Postman to contact the API to test direct interaction for each of the endpoints. The API definition is shown in Table 5.1.

5.2. Adding the API gateway

We configured KrakenD to expose the same interface as the API and to authenticate with it by adding the expected Authorization header before the request is sent by the gateway. The username password of the API was `test:test` that, encoded in base64, would generate the header `"Authorization: Basic dGVzdDp0ZXN0"`. We configured KrakenD using the `modifier/martian`[2] component to add the expected Authorization

[1]`https://go.dev/doc/tutorial/web-service-gin`
[2]`https://github.com/krakendio/krakend-martian`

Method	**Endpoint**	**Description**	**Parameters**	**Response**
GET	/users	Get all users	None	List of all users
GET	/users/{id}	Get user by ID	`{id}` - User ID	User information
POST	/users	Add a new user	User object	New user information
PATCH	/users/{id}	Update an existing user	`{id}` - User ID	Updated user information
DELETE	/users/{id}	Delete an existing user	`{id}` - User ID	Success message

Table 5.1: Sample API definition

```
{
  "endpoint": "/users",
  "method": "GET",
  "output_encoding": "json",
  "backend": [
      {
        "method": "GET",
        "host": [
          "http://host.docker.internal:5000"
        ],
        "is_collection": true,
        "url_pattern": "/users",
        "extra_config": {
            "modifier/martian": {
                "header.Modifier": {
                "scope": ["request"],
                "name": "Authorization",
                "value": "Basic dGVzdDp0ZXN0"
                }
            }
        }
      }
  ]
}
```

Listing 5.1: KrakenD `/users` endpoint configuration to let the API gateway act as a proxy

header automatically before contacting the backend.

We then tested the API was still working as intended by contacting it through the API gateway. The KrakenD configuration of a sample endpoint is shown in Listing 5.1.

To test the capabilities of KrakenD plugins, we also implemented a plugin for enabling basic auth validation at gateway level, since this feature was not available in the open-source version. The plugin validates that the Authorization header is the same as the expected, which is derived from the username and password field specified in its configuration. However, basic auth authentication is not completely secure as, even when using TLS which protect the credentials in transit, it always expose the username and password used to authenticate and this increase the attack surface. [2] A better approach would be to use access tokens, as presented in the following section.

5.3. Integrating with an IdP

Now that our API Gateway provides access to the target API, we can start adding security features to protect the API. In this section we show how we integrated KrakenD with an external IdP.

Since KrakenD is stateless the most viable approach to implement authentication and authorization features is to use JWT tokens because they are self-contained and therefore compatible with its architecture. In fact KrakenD cannot generate the tokens itself but has to rely on an external self-hosted or SaaS IdP that conforms to the OIDC standard and then be configured to validate those tokens [22]. In our work we tested how to integrate Auth0[3] as external SaaS IdP with KrakenD.

The focus of our work was not on the end users but on the API gateway, therefore we used the OAuth2.0 client credentials flow. This flow is used in machine-to-machine communication when the client is acting on his behalf, with no users involved. [8] We chose this flow because with this we could focus only on the role of API gateway in validating the token and we could automate the testing of the token generation and subsequent validation using command line tools. Also, the access token is the same independently of the flow, therefore even in scenarios when there are more complex flows with end users involved, the validation that the API gateway would do is the same.

Once the client application has received the access token, it can contact the API gateway providing the token inside the HTTP Authorization header of each request. The token is signed by Auth0 using RS256. The signing algorithm is asymmetric, therefore whoever has access to the public key can verify the token signature to prove authenticity, integrity and non-repudation [39], while only the signer have access to the private key that is used to sign the token. With this approach KrakenD only needs access to the public key from Auth0 to validate the tokens' signatures.

KrakenD does not do token introspection and only validates the tokens locally. The public key used to validate tokens signature can be cached to avoid contacting the IdP at every request: by default, KrakenD will check if the keys have changed every 15 minute, but it can be configured to change this setting if we want to rotate the keys periodically. [22].

[3] `https://auth0.com`

```
curl --location 'https://dev-5bkfe0he8u5v6zq3.uk.auth0.com/oauth/
    token' \
--header 'content-type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=client_credentials' \
--data-urlencode 'audience=AUDIENCE' \
--data-urlencode 'scope=read:users write:users' \
--data-urlencode 'client_id=CLIENT_ID' \
--data-urlencode 'client_secret=CLIENT_SECRET'
```

Listing 5.2: Curl request to get an access token from Auth0 using the client credentials flow

5.3.1. Generating JWT access tokens

For the first part, we configured Auth0 to grant access tokens for our API. Using postman, we crafted an HTTP request to require an access token , as shown in Listing 5.2, providing `client_id` and `client_secret` to authenticate, specifying the `grant_type` to indicate that we are using the OAuth2.0 client credentials flow [8], and the requested audience and scope.

Auth0 replies with a JWT token that contains the requested scopes, if the client application has been authorized with the requested ones. In our case we configured the requesting application to have permissions to read and write, using the custom scopes `read:users` and `write:users`.

5.3.2. Configuring KrakenD to protect endpoints

Listing 5.3 shows how to configure an endpoint in KrakenD to be protected with JWT. We need to specify at least the signing algorithm used to verify the tokens and the `"jwk_url"` parameter that is used to specify where KrakenD can get the public keys to verify the tokens' signatures. The `"audicence"` parameter is optional and rejects all tokens that do not include the audience specified inside the payload. The `"scopes"` parameter also validates that the JWT token has at least one of the scopes specified in the list. [22] We used the scope to distinguish between read and write authorization. In the former, the owner of the token can only read users, in the latter, the owner of the token can also perform write operations to create, update or delete users.

Since KrakenD force us to specify in the configuration the signing algorithm used, we avoid attacks that aim at removing the signature from the JWT changing the signing algorithm specified in the header `alg` to None, because KrakenD will only accept tokens

```
{
  "endpoint": "/users",
  "method": "GET",
  "output_encoding": "json",
  "extra_config": {
      "auth/validator": {
          "alg": "RS256",
          "jwk_url": "https://dev-5bkfe0he8u5v6zq3.uk.auth0.com/.
             well-known/jwks.json",
          "audience" : ["api://audience"],
          "scopes_key": "scope",
          "scopes_matcher": "any",
          "scopes": [
            "read:users"
          ],
          "cache": true
      }
  },
  ...
}
```

Listing 5.3: KrakenD configuration to enable JWT validation on an endpoint

signed with RS256.

5.3.3. Testing protected endpoints with Postman

Once we configured Auth0 to generate access tokens and KrakenD to validate them, we tested if the endpoints we wanted to protect were working as intended. We configured Postman to save the access token received from the first request as variable and add it as authorization header for subsequent requests. Then we verified that the endpoints were accessible with the token received, and not accessible without the token, or with an invalid token.

With Postman, we can automate these tests running scripts. We used this functionality to include tests that verified that a protected endpoint rejects tokens that are expired, or not present at all. We also tested that a correct token is actually accepted by the API gateway and forwarded for further processing. These tests can then be exported as a JSON file that could be later integrated inside a CI/CD pipeline using Newman, a tool that enable to run Postman collections from the command line.

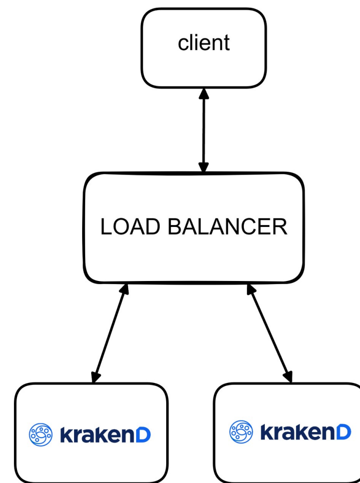


Figure 5.1: High availability cluster

5.4. Achieving high availability

Once we enabled authentication on our API, we tested how to achieve high availability.

5.4.1. Distributing ingress traffic with a load balancer

Since KrakenD is stateless, we can deploy multiple independent instances and distribute ingress traffic between them with a load balancer. With this approach, even if one instance fail, this does not imply that the entire system fails. [23]

We tested how to deploy a load balancer locally using Nginx and configure it to split the traffic among multiple KrakenD instances, using a round robin or weighted round robin algorithm. [37] In this case the only configuration that we need is to set up the load balancer, as KrakenD instances are independent and therefore does not need additional set up to operate in this scenario. [23] The architecture is schematized in Figure 5.1.

KrakenD developers recommend to have a couple of instances running even in low traffic environments for redundancy. [15]

5.4.2. Configuring mutual TLS between load balancer and KrakenD

In this section we show how we tested to configure TLS between the client and the load balancer, and mutual TLS between the load balancer and KrakenD instances, as schematized in Figure 5.2.

```
http {

    upstream upstream-server {

        server krakend1:port1;
        server krakend2:port2;

    }

    server {

        listen 80;

        location / {
            proxy_pass http://upstream-server;
        }
    }
}
```

Listing 5.4: Load balancer basic configuration to split traffic amongst KrakenD instances using a round robin balancing algorithm

First, we generated self-signed certificates since we were working locally. We generated a public/private key pair for the KrakenD instances, specified inside `/etc/krakend/tls/cert.pem` and `/etc/krakend/tls/key.pem`, as shown in Listing 5.6. We generated a public/private key pair for the load balancer and configured KrakenD to accept those in the mutual TLS configuration through the `"ca_certs"` array. Then, we enabled TLS authentication for the load balancer and mutual TLS for KrakenD instances. Finally, we configured the load balancer to enable mutual TLS authentication with KrakenD instances. The Nginx configuration sample in shown in Listing 5.5, while the KrakenD configuration snippet is shown in Listing 5.6.

5.4.3. Analyzing token revocation in a cluster

KrakenD does not do token introspection, it validates the JWT tokens locally. [22] In this way, it can not know whether a token has been revoked or not by asking the Identity Provider. Instead, each instance of the API gateway has its own bloomfilter which can be managed through RPC calls to revoke JWT tokens. A bloomfilter is a probabilistic data structure based on hashing that can tells us rapidly if an element is either definitely not in the set or may be in the set. [35]

The gateway read its local bloomfilter to check if a JWT token has been revoked. Based

```

http {

    upstream upstream-server {

        server krakend1:port1;
        server krakend2:port2;
    }


    server {

        listen              443 ssl;

        ssl_certificate     /etc/nginx/ssl/client.crt;
        ssl_certificate_key /etc/nginx/ssl/client.key;
        ssl_protocols       TLSv1.3;
        ssl_ciphers         HIGH:!aNULL:!MD5;

        location / {
            proxy_pass https://upstream-server;
            proxy_ssl_certificate         /etc/nginx/ssl/client.
               crt;
            proxy_ssl_certificate_key     /etc/nginx/ssl/client.
               key;
            proxy_ssl_protocols           TLSv1.3;
            proxy_ssl_ciphers             HIGH:!aNULL:!MD5;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

Listing 5.5: Nginx configuration to split traffic amongst KrakenD instances using a round robin balancing algorithm, with TLS from client to load balancer and mTLS between load balancer and KrakenD instances

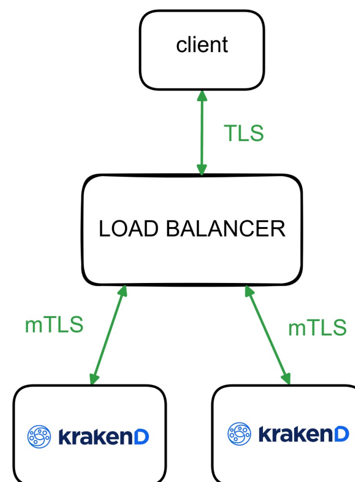


Figure 5.2: Load balancer architecture with TLS from client and mutual TLS with KrakenD instances

```
{
  "version": 3,
  "tls": {
    "public_key": "/etc/krakend/tls/cert.pem",
    "private_key": "/etc/krakend/tls/key.pem",
    "enable_mtls": true,
    "ca_certs": [
      "/etc/krakend/tls/mTLS/rootCA.pem"
    ]
  },
  ...
}
```

Listing 5.6: KrakenD sample config to enable mutualTLS

on the false positive rates a valid token could be rejected, but when configuring the bloomfilter, we can set a target false positive rate inside its configuration and it will be sized accordingly. The false positive rate can be reduced to an acceptable amount of one false positive every 10 million different tokens or even more [25].

This blacklist approach is implemented as a set of three sliding bloomfilters: previous, current and next.[4] An add operation, used to revoke a token or a group of tokens, is performed on the current and next bloomfilters, while a check operation, used to verify if a token or a group of token is revoked, is performed on previous and current. Periodically, the bloomfilters in the set are shifted: previous is deleted, the old current becomes previous and the old next becomes current, while next is created. This mechanism is leveraged to let a revoked token live inside the blacklist for a minimum of 2 x TTL, where TTL is the period, because with this approach we have a way to remove revoked tokens after they have expired, which is not trivial since the bloomfilter is based on hashing and being a one-way function we can't remove already revoked tokens once they are inserted inside the data structure. According to the documentation, the TTL parameter must be set the same as the expiry time of the tokens issued by the IdP: [25] this ensures that a token which is inserted into the bloomfilter is not removed before its actual expiration. Unfortunately the presented approach is not explained inside the documentation: the only thing we know from it is that the TTL must be set the same as the expire time of the tokens issued by the IdP. We decided to look at the code to understand this mechanisms because we thought it could be useful to have a more detailed overview on how this blacklist approach was actually implemented.

When we use a cluster of KrakenD instances, each instance need to receive revocation updates on its local bloomfilter. This can be achieved through a centralized revoker that propagate revocation updates and synchronize new instances that are added to the cluster or instances that crash and restart (that will lose all revocation updates since they are stored only in-memory). [25] To propagate an update we can use the add operation, while to synchronize the instances we can execute a union operation, which is performed on both previous, current and next. The advantage of the bloomfilter is that these operations are idempotent [29], so we don't need to ensure that the commands are executed exactly once but we just need at least once delivery, making the architectural requirements less complex.

We implemented a draft solution to test the feasibility of having a centralized revoke server, following a similar approach to the one used by KrakenD's developers in the

[4]`https://github.com/krakendio/bloomfilter/tree/master/rotate`

enterprise version[29], whose architecture is shown in Figure 5.6: the revoke server receives commands through a REST API and interact with each KrakenD instance with RPC, the instances self register to the centralized revoke server communicating an UUID used to distinguish them and the revoke server has its own bloomfilter to handle new or crashed instances with an union operation between the RPC client and server bloomfilters.

First, we implemented a simple REST API to receive commands to propagate to the instances. The API definition is synthetized in Table 5.4. We used the bloomfilter[5] component used by KrakenD developers in the gateway instances to have a local set of sliding bloomfilters inside the revoke server. This choice was made following the architecture of the centralized revoker in the enterprise version, because the centralized revoker not only has to propagate updates to the client, but also re-synchronize them in case of failure, as the bloomfilter of each KrakenD client is stored in memory. The revoke server mantain a list of active instances, and an RPC client[6] for each instance. When it receive an update operation, it propagates it through RPC using the functions provided by KrakenD's developers. Each instance bloomfilter is exposed through an RPC server.[7]

Then, we implemented a plugin to handle instances self-registration using a random UUID that is generated at startup. With this approach, the revoke server can know when an instance has crashed and restarted, since it will generate a new UUID. This approach cover cases in which KrakenD instantly crashes and restarts losing the status of its local bloomfilter: in this scenario using a simple ping from the revoker to the instance would not be able to determine if the instance has lost all the revocation updates, because based on the ping frequency the instance could still result as healthy. At startup, the instance will generate an UUID and keep sending it to the revoke server. This acts as a registration, if the instance is new or restarted from crash, or as a ping, if the instance is already registered. The ping from the plugin to the revoke server is used to stop accepting requests if the instance notices it lost connection to the centralized revoke server, since in this case it would possibly accept rejected tokens because its local bloom filter could not be updated with the last operations that were propagated by the revoke server.

Finally, we tested manually using Postman that the instances were correctly synchronized simulating crash restarts and new instances registration, ensuring that a revoked token was still revoked in the restarted instance or in the new instance. The strategy we used is synthetized in table 5.2.

The revoke server should be further improved to be used in a real scenario, starting by

[5]`https://github.com/krakendio/bloomfilter/`
[6]`https://github.com/krakendio/bloomfilter/blob/master/rpc/client/client.go`
[7]`https://github.com/krakendio/bloomfilter/blob/master/rpc/server/server.go`

Revoker helper (plugin)	Revoke server
• handles instance self registration to the revoke server • ping the revoke server • stop accepting requests if it loses connection to the revoke server	• handles token revocation propagation to the registered instances • ping the registered instances to keep an updated list of active ones • synchronize new/crashed instances with its bloom filter

Table 5.2: Centralized revoker strategy recap

serializing the server bloomfilter on disk and securing the communication between the instances and the revoker, both the communication of the instances to self-register and the RPC communication to interact with the bloomfilters. Since our goal was to test the feasibility of this solution, we only focused on the core aspects of centralizing token revocation realizing a draft solution that would give us an idea of the problems that arise in this approach, to see if it is worth it to handle it in this way.

We concluded that this limitation is tied to the stateless architecture of KrakenD and stateless JWT tokens: revocation in a cluster of instances increase the architectural complexity and we should reduce or avoid the need for it using short lived tokens and let them expire, possibly relying on the client to refresh them. If token revocation is necessary then an alternative solution could be to use a centralized bloomfilter instead of a decentralized one to check if the token has been revoked. This would reduce performances since at each JWT validation the instance would have to connect to the centralized bloomfilter to see if the token has been revoked, but it would eliminate the need for instance synchronization. Another possible approach would be to disable the caching features of the `auth/validator` component used to cache the public key needed to verify the token signature and rotate the public/private key pair. [13] With this approach we can invalidate all tokens instantaneously without the need to keep a centralized or decentralized blacklist since all the old signatures would instantly become invalid. However in this scenario we would put an increased burden on the identity provider as it would need to be contacted for validating the JWT signature on every request, and we could only invalidate all issued tokens altogether. Deciding which approach is better depends on the size of the cluster, the frequency of token revocation and the performance needs. Table 5.3 shows the differences between the approach of letting token expire versus using a blacklist versus changing the secret used to sign tokens. As we can see, the blacklist approach is immediate but increase the architectural complexity and scalability of our solution, as we saw by analyzing the cluster token revocation in KrakenD, in which this problem is even worsened by its stateless architecture.

Solution	Short lived	Black list	Secret change
Architectural complexity	Low	High	Low
Invalidation latency	Medium	Instant	Instant
Acquisition frequency	High	N/A	High
Scalability	Linear	Bad	Very bad

Table 5.3: Comparison of different JWT revocation methods, extracted from [13] and modified

REQ	PATH	PARAM	DESC
GET	/status		Used for health checks, returns HTTP status 200 if the revoke server is up and running.
GET	/instances		Return the list of active instances.
POST	/instances	"uuid" and "port" (RPC) in JSON request body	Add a new instance to the list.
POST	/tokens/{claim}/{key-to-revoke}		Revoke a token or a group of tokens.
GET	/tokens/{claim}/{key-to-revoke}		Check that a token or a group of tokens has been revoked in all active instances.

Table 5.4: Revoke server endpoints

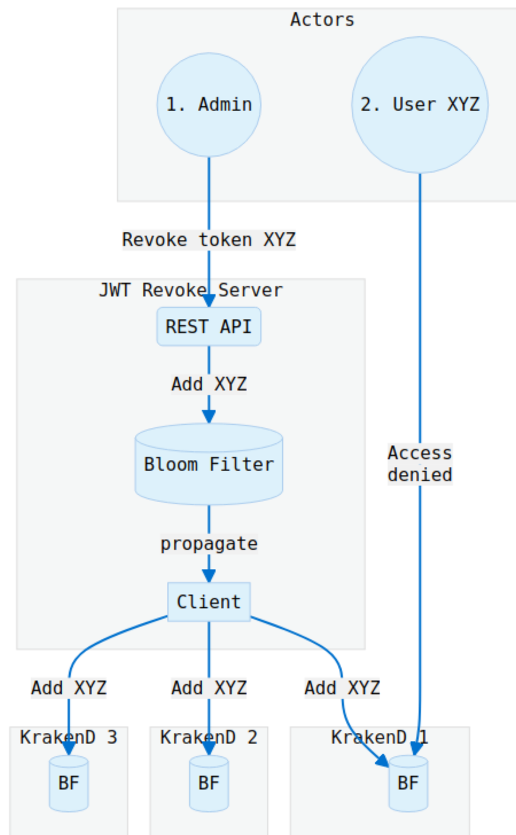
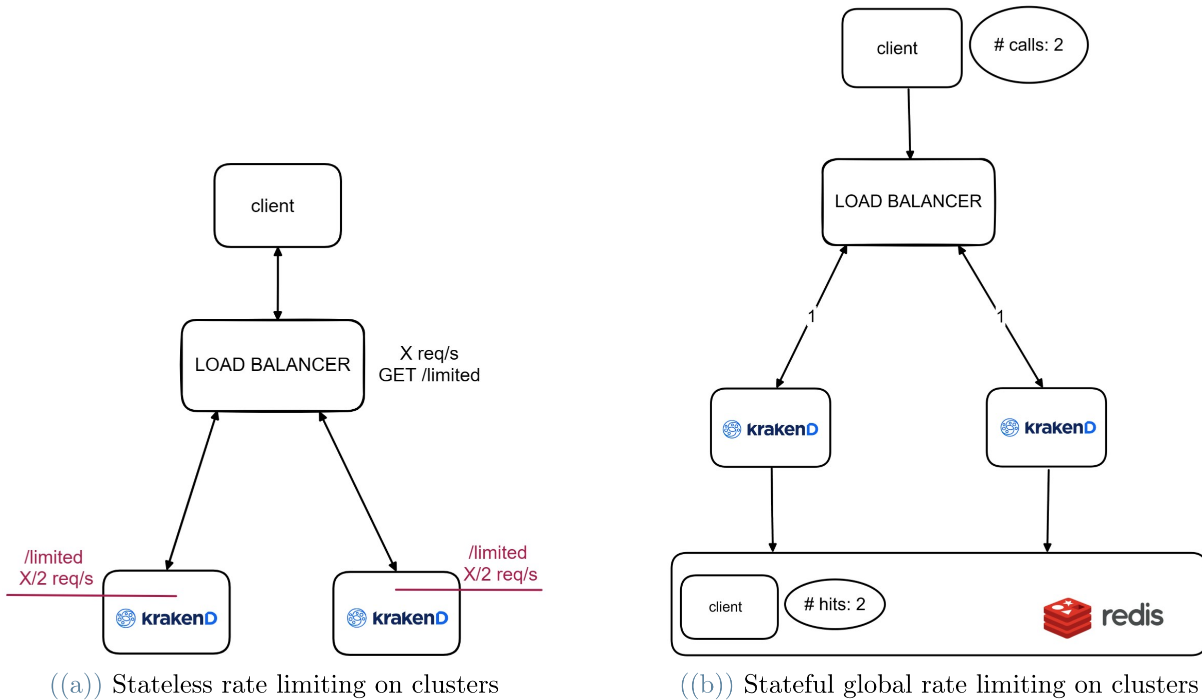


Figure 5.3: Centralized revoke server architecture, Extracted from [29]



((a)) Stateless rate limiting on clusters

((b)) Stateful global rate limiting on clusters

Figure 5.4: Rate limiting on clusters

5.5. Configuring rate limiting

As presented in chapter 4, we have two possible approaches to rate limiting in a cluster. The former is to split the total rate limits among the size of the cluster, the latter is to implement a custom plugin that relies on Redis to handle the rate limiting counters synchronization. We tested how to enable both of them in our solution, as shown in Figure 5.4.

5.5.1. Stateless rate limiting on clusters

To split the rate limit in a stateless cluster, we have to know the algorithm used by the load balancer. If we have a cluster of 2 KrakenD instances and a round robin balancing algorithm that split the traffic evenly across those and we want to achieve a total maximum rate of 10 reqs/minute for all user, we can set a `max_rate` of 5 reqs/minute for each instance. [32]

This approach is simple but the limit itself cannot be assured exactly since we don't have any guarantee that the traffic is going to be split evenly across the instances, even when using a round robin algorithm. In the case that one instance fails and all the traffic is temporarily redirected to the other one, our rate limit will be halved. This problem is

```
{
"endpoint": "/limited",
    "extra_config": {
      "qos/ratelimit/router": {
          "max_rate": 60,
          "client_max_rate": 20,
          "every": "1m",
          "strategy": "header",
          "key": "X-Real-IP"
        }
    },
    ...
}
```

Listing 5.7: KrakenD example configuration to enable both absolute rate limiting and client rate limiting on an endpoint, identified by different X-Real-IP header

even worse when we want to set a rate limit by client, identified either by IP address or HTTP header, since the round robin balancing algorithm does not take into account single clients but just forwards traffic in turns between the instances, therefore a single client, in the worst case, may be always redirected to only one node of the cluster and his actual rate limit will be halved.

We configured both a maximum rate limit on the endpoint and a rate limit differentiated by header X-Real-IP, which is the original IP of the caller forwarded by the load balancer as HTTP header. In this sample, shown in Listing 5.8 the endpoint is set with a maximum rate limit of 30 requests per minute and a rate limit by client of 10 requests per minute, identified by the X-Real-IP header forwarded by the load balancer.

5.5.2. Stateful global rate limiting on clusters

To address the limitation of the counters not synchronized in a cluster, we implemented a custom plugin that relies on Redis as in-memory database to introduce a shared state across multiple independent instances. Redis is suited for rate limiting because it has builtin commands that allows to handle counter access thread safe and set a time limit after which the key will automatically expire [47]. As an in-memory database it is faster than a traditional on-disk database, allowing us to limit the additional overhead requested by this approach.

We followed the same approach used by KrakenD developers, as they also offer their global cluster rate limit feature in the enterprise version [28].

```
{
"endpoint": "/limited",
    "extra_config": {
      "qos/ratelimit/router": {
          "max_rate": 30,
          "client_max_rate": 10,
          "every": "1m",
          "strategy": "header",
          "key": "X-Real-IP"
        }
    },
    ...
}
```

Listing 5.8: KrakenD example configuration for stateless rate limiting on clusters, splitting the traffic of the precedent configuration across two instances

The plugin is an HTTP handler (2.2.3) and therefore allows to intercept the request as soon as it hits the gateway, inspect the IP address of the request or a configurable HTTP header, check if the limit is reached and increment a shared counter. We chose to implement this plugin to set a global rate limiting at gateway level that address all requests on the gateway instances in our cluster, independently on the endpoints.

The rate limiting system we used is the leaky bucket, relying on an already existing Go library to handle the algorithm implementation[8]. When the request limit is reached, a 429 Too many requests HTTP error is sent to the client, as shown in Listing 5.9. The full plugin code is shown in Appendix A.

We chose to have a non-blocking rate limiter, meaning that if the connection with Redis is lost the limiting will be disabled and the plugin will try to reconnect to the database, while the gateway will continue accepting requests, even though it is also possible to have a blocking plugin that will turn down the gateway until the connection to the database is back. This choice was made to ensure that the employment of Redis does not introduce a new single point of failure in our architecture, since we can still use this rate limiting strategy in conjunction with the stateless mechanisms that we presented in the previous section.

We implemented a plugin that puts a global rate limit shared between all clients and a global rate limit for each client, identified by their IP address or by a configurable HTTP header which is extracted from the request.

[8] https://github.com/go-redis/redis_rate/

```
if res.Allowed == 0 {
    seconds := int(res.RetryAfter / time.Second)
    errorMessage := "Too Many Requests: retry after " + strconv.Itoa(
        seconds) + " seconds"
    http.Error(w, errorMessage, http.StatusTooManyRequests)
    return
} else {
    h.ServeHTTP(w, req)
    return
}
```

Listing 5.9: Global rate limit plugin logic

In our scenario we used the HTTP header, since when ingress traffic is distributed by a load balancer the original source IP address of the client will be replaced by the source IP address of the load balancer, but we can configure it to forward the original client IP inside an HTTP header.

We configured the plugin to allow to specify:

- **host.** The host where redis server resides.
- **port.** The port that redis server expose.
- **password.** The password used to authenticate the instance with the redis server.
- **header.** The HTTP header name we want to use as key in the redis database to differentiate clients.
- **rate.** The desired rate.
- **period.** The period in which the rate is considered. Can be `"minute"` or `"second"`.

  Note that the rate and period config must be the same across different instances to avoid inconsistencies.

The configuration of our plugin is sown in Listing 5.10.

5.5.3. Analyzing stateful and stateless rate limits

We can use both the presented rate limiting features in conjunction. The redis rate limiter provide a global rate limiting for each client, identified by their IP address which is forwarded by the load balancer as an HTTP header. We could also use the access token instead of the IP address to handle rate limiting. In that case we should ensure that the rate limiting windows is smaller than the token lifetime otherwise its accuracy would

```
"extra_config": {
    "plugin/http-server": {
      "name": ["global-rate-limit-header"],
      "global-rate-limit-header": {
            "host": "redis_host",
            "port": "redis_port",
            "password": "redis_password",
            "header": "X-Real-IP",
            "rate": "30",
            "period": "minute"
      }
    }
}
```

Listing 5.10: KrakenD example configuration to set up stateful global rate limit plugin, identifying client by the X-Real-IP HTTP header, with a rate limit of 30 requests per minute, to be placed as extra_config at root level

Solution	Stateless rate limit	Stateful rate limit
Architectural complexity	Lower	Higher
Accuracy	Lower	Higher
Performances	Faster	Slower
Scalability	Better	Worse

Table 5.5: Comparison of stateless and stateful rate limiting approaches

be decreased since the client would change the token and circumvent the limits. The endpoint limiter provide a limit for both each endpoint and each client on the endpoint, identified in the same way as the redis plugin.

Table 5.5 shows the comparison of the two approaches. The stateless rate limiting is faster but is not as accurate, as the instance would lose the counter state if restarted or crashed. The accuracy is further reduced when using a cluster because we don't have guarantees that the traffic is going to be split evenly by the load balancer, especially if we are rate limiting based on the IP address or by an HTTP header. The stateful rate limiting using an external database is slower but provides more accuracy as the instances counters are shared.

Overall, the advantage of using both the two approaches combined is that the requirements on the plugin that relies on redis can be lax since we can lose the connection to the database while still leveraging the local rate limits, and, at the same time, the plugin can act as a fallback for the case in which the instances are restarted and the standard rate

limiting counters lose their state, since they are stored only in memory. However, from a performance and scalability perspective relying on the default rate limiting counters would be better, when possible. [28]

5.6. Enabling monitoring tools

In this section we show how we tested the configurations to enable KrakenD to export logs, metrics and traces to external services.

First, we tested how to export logs to logstash [18]. KrakenD developers provide a preconfigured ELK stack[9] dashboard that contains all the necessary tools to process the logs and visualize them once exported. To configure KrakenD to export access and application logs we first need to enable logging and then export logs by enabling `telemetry/logging`[10] and `telemetry/gelf`[11] components in the configuration file [18], as shown in listing 5.11. The logs are sent over UDP to logstash and can be also sent over TCP, but for performance reasons UDP is recommended [18]. The library used by KrakenD developers to export logs does not support TLS[12] or DTLS. In this case the usual solution is to have a collector in the local machine that runs the instance, which is responsible to send logs over a secure channel. Securing the communication between KrakenD and monitoring tool was out of our scope, so we did not analyze these scenarios.

Then, we tested how to export metrics to an InfluxDB time series database and visualize them in a dashboard. Also in this case KrakenD developers provide a sample Grafana dashboard that is configured to visualize live information about the API gateway usage, reading the data from the InfluxDB database. The configuration 5.12 shows how to [20] :

1. enable the extended metrics to collect the information
2. push the data collected to influxdb

Finally, we tested how to export traces to Jaeger[13], to reconstruct the end-to-end traffic flow of requests. The configuration 5.13 shows how to export traces to a Jaeger collector.

We deployed all the monitoring tools using docker compose to test that KrakenD was correctly exporting data to the external services.

---

[9] `https://www.elastic.co/what-is/elk-stack`
[10] `https://github.com/krakendio/krakend-gologging`
[11] `https://github.com/krakendio/krakend-gelf`
[12] `https://github.com/Graylog2/go-gelf/issues/39`
[13] `https://github.com/jaegertracing/jaeger`

```
1  "extra_config": {
2      "telemetry/logging": {
3        "level": "WARNING",
4        "@comment": "Prefix should always be inside [] to keep the
            grok expression working",
5        "prefix": "[KRAKEND]",
6        "syslog": false,
7        "stdout": true
8      },
9      "telemetry/gelf": {
10       "address": "logstash:12201",
11       "enable_tcp": false
12     }
13 }
```

Listing 5.11: KrakenD config to export logs to logstash as extra_config to be placed at root level, extracted from [18]

```
1  "extra_config": {
2    "telemetry/influx":{
3        "address": influxdb:port",
4        "ttl":"25s",
5        "buffer_size":0,
6        "db": "bucket_name",
7        "username": "influx_user",
8        "password": "influx_password"
9    },
10   "telemetry/metrics": {
11     "collection_time": "30s",
12     "listen_address": "127.0.0.1:8090"
13   }
14 }
```

Listing 5.12: KrakenD example configuration to export metrics to influxdb, as extra_config to be placed at root level, extracted from [20]

```
"extra_config": {
    "telemetry/opencensus": {
      "sample_rate": 100,
      "reporting_period": 0,
      "exporters": {
        "jaeger": {
          "endpoint": "jaeger:14268/api/traces",
          "service_name":"krakend",
          "buffer_max_count": 1000
        }
      }
    },
}
```

Listing 5.13: KrakenD example configuration to export traces to Jaeger as extra_config to be placed at root level, extracted from [21]

5.7. Analyzing the deployment phase

In this section we analyze aspects related to the deployment phase of KrakenD.

5.7.1. Injecting secrets into the configuration file

Since the configuration file should be versioned, it is recommended not to put credentials inside the version control systems, even if they are private, as it pose the risk of exposing them. Instead we should manage secrets externally: in this way we can securely store credentials using secrets management tools that provide encryption at rest and access control policies. [5]

In our case we tested the KrakenD flexible config[14] that uses Go template package[15] allowing to specify variables for the secrets that needs to be inside the configuration file, and dynamically loads them from files or environment variables. The secrets in our scenario are injected into the pod by Kubernetes secrets.

We chose to inject files instead of environment variables as the latter are more easily exposed [38] [49]. Also, secrets mounted inside pods with Kubernetes Secrets are injected in tmpfs mounts that are only persisted in memory on the host machine [33]. Listing 5.14 shows how to inject the redis password used by the plugin to authenticate with the redis server loading it from a file that is stored in the path `/etc/krakend/secrets/redis/password`.

[14] `https://www.krakend.io/docs/configuration/flexible-config/`
[15] `https://pkg.go.dev/text/template`

```
1   "extra_config": {
2     "plugin/http-server": {
3       "name": ["global-rate-limit"],
4       "global-rate-limit": {
5               "host": "redis_host",
6               "port": "redis_port",
7               "password": "{{ include "/etc/krakend/secrets/redis/
                   password" }}",
8               "rate": "1",
9               "period": "minute"
10      }
11    }
12  }
```

Listing 5.14: KrakenD example configuration using Go templates to dinamically inject redis password into the configuration from a file

We could improve this mechanism by obliterating the secrets once they are consumed, as KrakenD only needs to read the configuration file at startup. However, this approach requires more work as we need a way to check that KrakenD has loaded before deleting the secrets, and reinject it in case of crashes and restart.

The full configuration with the features analyzed in the previous section with secrets loaded from files using this approach is shown in appendix A.

5.7.2. Implementing a CI/CD pipeline

In this section we present how we containerized KrakenD along with the custom plugins and how we implemented a Jenkins CI/CD pipeline to automatically build, test and deploy the gateway on Kubernetes. If we use containers, KrakenD's developers recommend to use an immutable infrastructure which consists of immutable Docker images with the configuration file embedded in them, managed by a CI/CD pipeline and orchestrated by Kubernetes [15].

First, we containerized KrakenD along with our custom plugins. Starting from the Dockerfile[16] of the official KrakenD repository, we modified it to add our plugins and specify the configuration file as build argument. The full Dockerfile is shown in appendix A.

Then, we implemented the CI/CD pipeline that does the following: pulls our repository from git that contains the plugins, krakend and the configuration file, builds the Dockerfile while doing build tests, push the image to a private registry, deploy it on Kubernetes and

[16] https://github.com/krakendio/krakend-ce/blob/master/Dockerfile

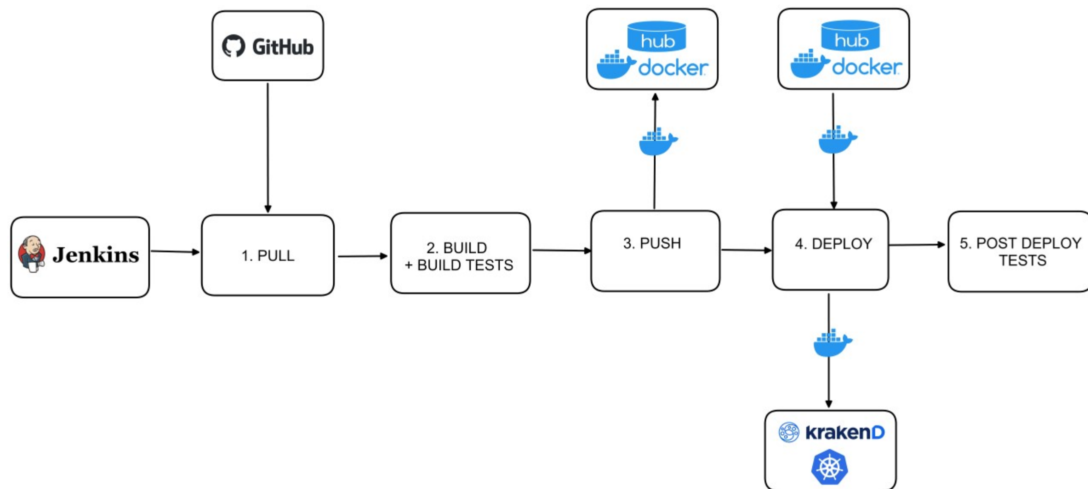


Figure 5.5: Jenkins CI/CD pipeline structure

lastly execute post deploy tests. The pipeline structure is shown in Figure 5.5. During the build phase we included the test of the configuration file syntax using the check tool[17] and the check of the configuration file for security misconfigurations using the audit tool[18]. Both tools are bundled in KrakenD.

After the deployment, we configured Jenkins to execute Postman collections tests from command-line using Newman[19]. In this way we can execute Postman tests authenticating with the external IdP and testing the protected endpoints automatically.

5.7.3. Configuring a secure network architecture

Even though in our environment we did not test this part directly, we analyzed the high level approach to configure a secure network architecture when using a cluster or KrakenD instances. In this scenario we should expose only the load balancer in the DMZ and then put all the other services inside the private network, setting up two firewalls. If KrakenD instances are deployed in different physical networks, we can setup a virtual private network. A possible approach to configure our network architecture, assuming we are using stateful packet filters, is listed in Table 5.6 and shown in Figure 5.6. If we want to enable token revocation using the centralized revoke server that we presented in Chapter 5, we should take into account that the revoke server communicate to the KrakenD instances over an RPC port which is not secured. If the external services used are outside the private network, then we should also add rules to ensure that each gateway

[17] https://www.krakend.io/docs/configuration/check/
[18] https://www.krakend.io/docs/configuration/audit/
[19] https://learning.postman.com/docs/collections/using-newman-cli/command-line-integration-with-newman/

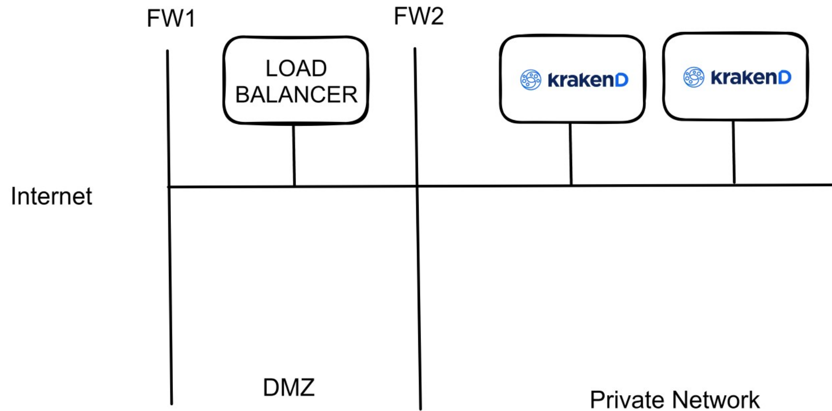


Figure 5.6: Cluster recommended network architecture

FW	src IP	src PORT	direction of 1st packet	dst IP	dst PORT	Policy
FW1	*	*	www->DMZ	*	*	Deny
FW1	*	*	www->DMZ	LB_IP	443	Allow
FW1	KrakenD_1_IP	*	DMZ->www	*	443	Allow
FW1	KrakenD_2_IP	*	DMZ->www	*	443	Allow
FW2	*	*	DMZ->private net	*	*	Deny
FW2	LB_IP	*	DMZ->private net	KrakenD_1_IP	KrakenD_1_Port	Allow
FW2	LB_IP	*	DMZ->private net	KrakenD_2_IP	KrakenD_2_Port	Allow
FW2	KrakenD_1_IP	*	private net->DMZ	*	443	Allow
FW2	KrakenD_2_IP	*	private net->DMZ	*	443	Allow

Table 5.6: Sample firewall rules for stateful packet filter

instance can contact those.

5.7.4. Benchmarking the gateway overhead

We used Hey[20], a simple command-line tool written in Go, to generate HTTP load to a single gateway endpoint testing it on our local environment, to have an idea of the overhead generated by adding each functionality starting from the base case, in order to understand which functionalities were heavier. The base case is calling the gateway health endpoint with no authentication and no additional functionality. In this scenario there is no target API involved. From this we added functionalities like authentication, observability tools integration and rate limiting configurations with an high limit to see how much the request processing capabilities of the gateway are reduced by adding these additional functionalities. Listing 5.15 shows the command that we used for the two local tests. The -c and -n flag controls the number of workers to run concurrently and the number of requests to run. We tested each different configuration with the same parameters to test the time that the gateway takes to process 100000 requests.

[20] `https://github.com/rakyll/hey`

```
1 hey -c 100 -n 100000 gateway:port/endpoint
```

Listing 5.15: Hey local tests commands

Auth	Rate Limit	Observability	total (s)	max (s)	min (s)	avg (s)	rps	200
-	-	-	1.1132	0.0112	0.0001	0.0011	89832.3451	100000
Basic Auth	-	-	11.4525	0.0938	0.0002	0.0113	8731.7093	100000
JWT	-	-	4.4865	0.0697	0.0003	0.0044	22288.9862	100000
-	-	Yes	6.8764	0.0786	0.0003	0.0067	14542.4659	100000
Basic Auth	-	Yes	60.7048	0.3333	0.0004	0.0598	1647.3163	100000
JWT	-	Yes	17.3414	0.1361	0.0007	0.0171	5766.5409	100000
-	Redis Limit	-	52.5816	0.1102	0.0102	0.0526	1901.8077	100000
Basic Auth	Redis Limit	-	112.1676	0.1790	0.0244	0.1121	891.5229	100000
JWT	Redis Limit	-	54.1057	0.3843	0.0053	0.0541	1848.2328	100000
-	Redis Limit	Yes	87.2537	0.5548	0.0025	0.0872	1146.0833	100000
Basic Auth	Redis Limit	Yes	166.4514	0.7228	0.0067	0.1664	600.776	100000
JWT	Redis Limit	Yes	175.9932	1.1433	0.0047	0.1759	568.2036	100000
-	Endpoint Limit	-	2.2675	0.1896	0.0001	0.0022	44102.1431	100000
Basic Auth	Endpoint Limit	-	3.8363	0.0247	0.0002	0.0038	26066.9802	100000
JWT	Endpoint Limit	-	4.7137	0.0531	0.0003	0.0046	21214.5395	100000
-	Endpoint Limit	Yes	16.0792	0.1137	0.0006	0.0158	6219.1996	100000
Basic Auth	Endpoint Limit	Yes	16.1453	0.1077	0.0006	0.0159	6193.7484	100000
JWT	Endpoint Limit	Yes	18.2499	0.1125	0.0008	0.0180	5479.4757	100000

Table 5.7: Local tests with Hey

The results of these tests are listed in Table 5.7. We saw that the additional functionalities decrease the processing capabilities of the gateway, especially the observability tools since every single request is logged by the access logs, reducing the ability of the gateway to handle requests concurrently. We also confirmed that, unsurprisingly, the stateless rate limiting approach is faster with respect to using redis for rate limiting with a plugin.

We also tested the time difference between calling the sample API directly from the load balancer and putting the API gateway in between, as shown in Table 5.8, to have an idea of the overhead without the network to validate a JWT token, handle rate limit, logging and request routing. In this case we used the sample API which we introduced in chapter 5. The overhead in our local machine was around 250 ms to process 200 requests. In this test we used less requests because we wanted to test the overhead under normal usage.

Scenario	Req	total (s)	max (s)	min (s)	avg (s)	rps	200
LB - API	GET /users	0.1789	0.1765	0.1076	0.1626	1117.9569	200
LB - API GW - API	GET /users	0.4375	0.3234	0.0071	0.0842	457.1428	200
LB - API	GET /users/1	0.2287	0.2261	0.1834	0.2111	874.4975	200
LB - API GW - API	GET /users/1	0.4822	0.3471	0.0057	0.0918	414.7656	200

Table 5.8: Local tests with Hey with load balancer

5.7.5. Load testing the API gateway

With Gatling[21], a more structured load testing tool developed in Scala, we tested how to simulate users on different endpoints of the API GW to reproduce realistic scenarios under normal and heavy traffic, analyzing how many requests the cluster can handle before it reaches its breaking point and starts failing to serve some requests. In our case we considered the breaking point when the gateway started to fail a small percentage of requests, even if the majority of requests could still be processed. We configured Gatling to generate access tokens from Auth0 and subsequently use them for the protected endpoints.

Similarly to the previous approach, we used the gatway health endpoint to test the processing capabilities of the API gateway without the target API. In this way, we could focus on testing only the capabilities of the API gateway directly before placing it in front of the services it needs to expose. In this case, we tested the cluster with the load balancer and one instance locally and 2 more instances on different cloud providers.

For the purpose of load testing the cluster to find the breaking point, we used plain HTTP connections. Depending on the scenario, we may want to enable TLS between the client and the load balancer and between the load balancer and the instances of the cluster. We executed load tests that simulate a peak of traffic on the span of 30 seconds on the endpoints listed in Table 5.9.

As shown in Figure 5.7 and 5.8, we found the breaking point of our cluster, between 2200 and 2300 users, for a total of 11000 and 11500 requests, meaning that in the simulations that generate 11500 requests or more the KrakenD instances composing the cluster starts cannot process all requests and start to reject some of them returning HTTP 500 status codes. The results were less than what we expected, mainly for three motives: the first is that having the load balancer locally may have introduced a new bottleneck in the architecture, since in the local load tests with hey we called the gateway directly without the load balancer, the second is that with respect to the local tests now we also have the network that could act as a bottleneck, the third is that the cloud instances had less resources than my local machine and could have introduced a new bottleneck with respect to the instance deployed locally.

The focus of our work was not on the results of the tests that vary on different environments but rather on the process of load testing the API gateway to dimension it correctly according to the traffic we expect to receive. Once we set up a cluster we can execute load tests and possibly scale our solution:

[21] `https://github.com/gatling/gatling`

REQ	ENDPOINT	AUTH	RATE LIMIT
GET	/limited	-	Endpoint Rate Limit
GET	/protected/bauth	Basic Auth	-
GET	/limited/protected/bauth	Basic Auth	Endpoint Rate Limit
GET	/protected/jwt	JWT	-
GET	/limited/protected/jwt	JWT	Endpoint Rate Limit

Table 5.9: Gatling simulation endpoints requests for each simulated user

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	11005	11005	0	0%	65.898	2	151	1202	15171	65143	86278	3915	13542
Authenticate	5	5	0	0%	0.03	296	341	352	1467	1690	1746	607	570
Health Endpoint Limited	2200	2200	0	0%	13.174	2	135	772	7319	65064	86089	2509	10381
Basic Auth Protected Endpoint	2200	2200	0	0%	13.174	2	144	1187	15133	65134	86159	3375	12017
Basic Auth Protected Limited Endpoint	2200	2200	0	0%	13.174	2	149	1259	21029	65157	86278	3938	13621
Protected Endpoint	2200	2200	0	0%	13.174	2	216	1295	64699	65158	84987	4632	14858
Protected and Limited Endpoint	2200	2200	0	0%	13.174	2	189	1248	64741	65126	65294	5128	15956

Figure 5.7: Gatling simulation before breaking point

- **horizontally.** by adding more instances to the cluster.
- **vertically.** by increasing the resources of each instance composing the cluster.

In both cases we should also take care that the load balancer and the other external services employed are scaled correctly to handle the increased traffic.

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	11505	11126	379	3%	74.708	2	136	1203	29955	65207	87271	5110	14246
Authenticate	5	5	0	0%	0.032	320	340	359	452	470	475	365	56
Health Endpoint Limited	2300	2268	32	1%	14.935	2	136	687	18959	65065	65389	2559	9740
Basic Auth Protected Endpoint	2300	2200	100	4%	14.935	2	137	1196	24592	65131	86591	4181	12442
Basic Auth Protected Limited Endpoint	2300	2202	98	4%	14.935	2	142	1290	29750	65215	86778	5510	14566
Protected Endpoint	2300	2230	70	3%	14.935	2	133	1334	64771	65478	86899	6679	16858
Protected and Limited Endpoint	2300	2221	79	3%	14.935	2	127	1459	64676	65222	87271	6633	16024

Figure 5.8: Gatling simulation after breaking point

6 | Conclusions

In this chapter we provide a recap of the advantages and limitations that we found about KrakenD through all our work and we conclude by analyzing possible future works that can be done starting from what we did.

In our work we analyzed the role of API gateways in the field of API security. We demonstrated that their employment is beneficial for the overall API security posture and that they are suited for protecting and exposing APIs, even though they provide foundational security features and should be used in conjunction with other advanced security tools.

We showed the steps that are needed to configure KrakenD for common scenarios, including: route requests to an existing API, integrate with an Identity Provider and validate its access tokens, configure rate limiting mechanisms, use multiple instances to achieve high availability, export information to monitoring tools and analyze aspects related to the deployment.

We found both advantages and limitations that comes with its stateless architecture.

For the advantages, we can easily scale horizontally, we can achieve high availability with a cluster of multiple independent instances where the failure of one instance does not compromise the overall system availability, we can version the configuration file to manage all the configurations smoothly and we can leverage immutable docker images that allows deployments to be safer and simpler. [9]

For the limitations, we analyzed the difficulties of handling token revocation in a cluster and concluded that using short lived tokens to avoid the need for token revocation is a better approach, when possible. We studied the inconsistencies generated by having rate limiting counters stored in memory and not synchronized across instances. We showed how relying on external services can be beneficial but can also introduce a new single point of failure in the architecture. However, some functionalities like generating tokens are needed somewhere in the system. The immutable infrastructure approach has its advantages but requires redeployment to apply changes to the configuration, whereas in a

stateful gateway this changes could be less tedious to achieve. However, we showed that this limitation can be mitigated by using a DevOps approach.

We can use this work as a starting point to evaluate if KrakenD is suited for a specific business scenario. In the following section we present some future works that can be done.

For authentication and authorization, we could implement a plugin to allow API keys. The API Key inbound validation is not present in the open-source version of KrakenD. Its stateless architecture require additional work to implement this mechanism, since for every API key we want to allow we must re-deploy the gateway. One possible approach to address this limitation, used by KrakenD developers in the enterprise version, is to ship the gateway with some pre-loaded API keys in the configuration file and manage the grant of those keys using an external backend service that keeps track of the correlation between API key and user which the key it has been assigned to [27].

We could improve the architecture by using tools to automatically scale up or down the size of the cluster based on the current load. The load balancer should be deployed on a dedicated high performance machine as it can easily become the bottleneck in the architecture.

The cluster token revocation is tedious and depending on the scenario we would consider using a different approach other than the one we presented if token revocation is critical for that scenario, or improve the centralized revocation system. By using short-lived token we can reduce the need for token revocation making this less of a problem.

The open source version of KrakenD does not have a global rate limiter. Implementing it and splitting the limits amongst the cluster size instead of relying on an external redis database would be a viable solution to keep the performance advantages of the stateless architecture, paying with less accuracy. If possible, using the stateless rate limiting approach is better since the performance advantages of using local counters without instance synchronization outweighs the accuracy disadvantages. However, if we constantly scale the cluster size having shared rate limiting counters in a database would be a better solution.

Another possible future work is to analyze in depth the input validation mechanisms as in our work we did not focus on this part.

Bibliography

[1] Auth0. Access token, 2023. URL https://auth0.com/docs/secure/tokens/access-tokens.

[2] AviD. Is basic-auth secure if done over https?, 2023. URL https://security.stackexchange.com/questions/988/is-basic-auth-secure-if-done-over-https.

[3] V. Bertocci. On the nature of oauth2's scopes, 2023. URL https://auth0.com/blog/on-the-nature-of-oauth2-scopes/.

[4] Cloudflare. What is api security?, 2023. URL https://www.cloudflare.com/learning/security/api/what-is-api-security/.

[5] d33tah. Why is storing passwords in version control a bad idea?, 2023. URL https://security.stackexchange.com/questions/191590/why-is-storing-passwords-in-version-control-a-bad-idea.

[6] D. Docs. Runtime options with memory, cpus, and gpus, 2023. URL https://docs.docker.com/config/containers/resource_constraints.

[7] R. Documentation. Api security, 2023. URL https://www.redhat.com/en/topics/security/api-security.

[8] I. E. T. Force. The oauth 2.0 authorization framework, 2023. URL https://www.rfc-editor.org/rfc/rfc6749.

[9] Google. Best practices for operating containers, 2023. URL https://cloud.google.com/architecture/best-practices-for-operating-containers.

[10] R. Hat. What does an api gateway do?, 2023. URL https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do.

[11] R. Hat. What are apis, 2023. URL https://www.redhat.com/en/topics/api.

[12] I. C. E. IBM Cloud Education. What are api gateways?, 2023. URL https://www.ibm.com/cloud/blog/api-gateway.

[13] L. V. Jánoky, J. Levendovszky, and P. Ekler. An analysis on the revoking mechanisms for json web tokens. *International Journal of Distributed Sensor Networks*, 14(9), 2018. doi: 10.1177/1550147718801535.

[14] KrakenD. Understanding the configuration file, 2023. URL `https://www.krakend.io/docs/configuration/structure/`.

[15] KrakenD. Production best practices, 2023. URL `https://www.krakend.io/docs/deploying/#deployment-recommendations`.

[16] KrakenD. Docs - krakend api gateway, 2023. URL `https://www.krakend.io/docs/`.

[17] KrakenD. Conditional requests and responses with cel, 2023. URL `https://www.krakend.io/docs/endpoints/common-expression-language-cel/`.

[18] KrakenD. Elk stack dashboard, 2023. URL `https://www.krakend.io/docs/logging/elk-integration/`.

[19] KrakenD. Flexible configuration: template-based config, 2023. URL `https://www.krakend.io/docs/configuration/flexible-config/`.

[20] KrakenD. Exporting metrics and events to influxdb, 2023. URL `https://www.krakend.io/docs/telemetry/influxdb/`.

[21] KrakenD. Exporting traces to jaeger, 2023. URL `https://www.krakend.io/docs/telemetry/jaeger/`.

[22] KrakenD. Json web token validation, 2023. URL `https://www.krakend.io/docs/authorization/jwt-validation/`.

[23] KrakenD. Load balancing, 2023. URL `https://www.krakend.io/docs/throttling/load-balancing/`.

[24] KrakenD. Router rate-limiting, 2023. URL `https://www.krakend.io/docs/endpoints/rate-limit/`.

[25] KrakenD. Revoking valid tokens with a bloom filter, 2023. URL `https://www.krakend.io/docs/authorization/revoking-tokens/`.

[26] KrakenD. Understanding the token bucket algorithm, 2023. URL `https://www.krakend.io/docs/throttling/token-bucket/`.

[27] KrakenD. Api key authentication with rate limiting, 2023. URL `https://www.krakend.io/docs/enterprise/authentication/api-keys/#protecting-endpoints-by-api-key`.

[28] KrakenD. Redis-based global rate limit (stateful), 2023. URL https://www.krakend.io/docs/enterprise/throttling/global-rate-limit/.

[29] KrakenD. Revoke server for clusters, 2023. URL https://www.krakend.io/docs/enterprise/authentication/revoke-server/.

[30] Krakend. Introduction to custom plugins and middlewares, 2023. URL https://www.krakend.io/docs/extending/.

[31] KrakenD. Understanding the benefits of an stateless api gateway, 2023. URL https://www.krakend.io/blog/benefits-of-stateless-api-gateway/.

[32] KrakenD. Stateless rate-limiting on clusters, 2023. URL https://www.krakend.io/docs/throttling/cluster/.

[33] Kubernetes. Secrets, 2023. URL https://kubernetes.io/docs/concepts/configuration/secret/.

[34] R. Li. The past, present, and future of api gateways, 2023. URL https://www.infoq.com/articles/past-present-future-api-gateways/.

[35] llimllib. Bloom filters by example, 2023. URL https://llimllib.github.io/bloomfilter-tutorial/.

[36] Microsoft. Recommendations to mitigate owasp api security top 10 threats using api management, 2023. URL https://learn.microsoft.com/en-us/azure/api-management/mitigate-owasp-api-threats.

[37] Nginx. What is round-robin load balancing?, 2023. URL https://www.nginx.com/resources/glossary/round-robin-load-balancing/.

[38] nillsf. Don't use environment variables in kubernetes to consume secrets, 2023. URL https://blog.nillsf.com/index.php/2020/02/24/dont-use-environment-variables-in-kubernetes-to-consume-secrets/.

[39] O'Reilly. National institute of standards and technology, 2023. URL https://csrc.nist.gov/glossary/term/digital_signature.

[40] O'Reilly. The three pillars of observability, 2023. URL https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html.

[41] OWASP. Owasp api security project, 2023. URL https://owasp.org/www-project-api-security/.

[42] OWASP. Owasp top 10 api security risks – 2019, 2023. URL https://owasp.org/API-Security/editions/2019/en/0x11-t10/.

[43] OWASP. Owasp top 10 api security risks – 2023, 2023. URL https://owasp.org/API-Security/editions/2023/en/0x11-t10/.

[44] OWASP. Rest security cheat sheet, 2023. URL https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html.

[45] D. Puzder. Vulnerabilities, threats, and risks explained, 2023. URL https://informationsecurity.wustl.edu/vulnerabilities-threats-and-risks-explained/.

[46] RedHat. Stateful vs stateless, 2023. URL https://www.redhat.com/it/topics/cloud-native-apps/stateful-vs-stateless.

[47] Redis. Rate limiting, 2023. URL https://redis.com/glossary/rate-limiting.

[48] S. Security. State of api security q1 2023, 2023. URL https://salt.security/api-security-trends.

[49] Tenable. 5.4.1 prefer using secrets as files over secrets as environment variables, 2023. URL https://www.tenable.com/audits/items/CIS_Kubernetes_v1.20_v1.0.0_Level_2_Master.audit:98de3da69271994afb6211cf86ae4c6b.

[50] Verizon. Data breach investigations report, 2023. URL https://www.verizon.com/business/resources/reports/dbir/.

[51] E. Wittern. Web apis - challenges, design points, and research opportunities: Invited talk at the 2nd international workshop on api usage and evolution (wapi '18). In *2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*, pages 18–18, 2018.

A | Appendix

```
{
  "$schema": "https://www.krakend.io/schema/v2.4/krakend.json",
  "version": 3,
  "plugin": {
    "pattern": ".so",
    "folder": "/usr/bin/plugins"
  },
  "tls": {
    "public_key": "/etc/krakend/tls/cert.pem",
    "private_key": "/etc/krakend/tls/key.pem",
    "enable_mtls": true,
    "ca_certs": [
      "/etc/krakend/tls/mTLS/rootCA.pem"
    ],
    "disable_system_ca_pool": false
  },
  "name": "krakend-config-sample-final",
  "output_encoding": "json",
  "port": "{{ env "KRAKEND_PORT" }}",
  "endpoints": [
    {
      "endpoint": "/users",
      "input_headers": ["X-Real-IP"],
      "method": "GET",
      "output_encoding": "json",
      "extra_config": {
          "auth/validator": {
              "alg": "RS256",
              "audience": ["api://audience"],
              "jwk_url": "https://dev-5bkfe0he8u5v6zq3.uk.auth0.
                 com/.well-known/jwks.json",
```

```
31              "scopes_key": "scope",
32              "scopes_matcher": "any",
33              "scopes": [
34                "read:users"
35              ],
36              "cache": true
37          },
38          "qos/ratelimit/router": {
39            "max_rate": 5,
40            "client_max_rate": 5,
41            "every": "1m",
42            "strategy": "header",
43            "key": "X-Real-IP"
44        }
45      },
46      "backend": [
47          {
48            "method": "GET",
49            "host": [
50              "http://host.docker.internal:5000"
51            ],
52            "is_collection": true,
53            "url_pattern": "/users",
54            "extra_config": {
55              "modifier/martian": {
56              "header.Modifier": {
57              "scope": ["request"],
58              "name": "Authorization",
59              "value": "{{ include "/etc/krakend/secrets/basic-
                   auth" }}"
60              }
61             }
62            }
63          }
64      ]
65    },
66    {
67      "endpoint": "/users/{user}",
68      "input_headers": ["X-Real-IP"],
```

```
69        "method": "GET",
70        "output_encoding": "json",
71        "extra_config": {
72            "auth/validator": {
73                "alg": "RS256",
74                "audience": ["api://audience"],
75                "jwk_url": "https://dev-5bkfe0he8u5v6zq3.uk.auth0.
                     com/.well-known/jwks.json",
76                "scopes_key": "scope",
77                "scopes_matcher": "any",
78                "scopes": [
79                  "read:users"
80                ],
81                "cache": true
82            },
83            "qos/ratelimit/router": {
84              "max_rate": 5,
85              "client_max_rate": 5,
86              "every": "1m",
87              "strategy": "header",
88              "key": "X-Real-IP"
89          }
90        },
91        "backend": [
92            {
93              "method": "GET",
94              "host": [
95                "http://host.docker.internal:5000"
96              ],
97              "url_pattern": "/users/{user}",
98              "extra_config": {
99                "modifier/martian": {
100               "header.Modifier": {
101               "scope": ["request"],
102               "name": "Authorization",
103               "value": "{{ include "/etc/krakend/secrets/basic-
                     auth" }}"
104               }
105              }
```

```
106              }
107            }
108        ]
109      },
110      {
111        "endpoint": "/users",
112        "input_headers": ["X-Real-IP"],
113        "method": "POST",
114        "output_encoding": "json",
115        "extra_config": {
116            "auth/validator": {
117                "alg": "RS256",
118                "audience": ["api://audience"],
119                "jwk_url": "https://dev-5bkfe0he8u5v6zq3.uk.auth0.
                      com/.well-known/jwks.json",
120                "scopes_key": "scope",
121                "scopes_matcher": "any",
122                "scopes": [
123                  "write:users"
124                ],
125                "cache": true
126            },
127            "qos/ratelimit/router": {
128              "max_rate": 5,
129              "client_max_rate": 5,
130              "every": "1m",
131              "strategy": "header",
132              "key": "X-Real-IP"
133          }
134        },
135        "backend": [
136            {
137              "method": "POST",
138              "host": [
139                "http://host.docker.internal:5000"
140              ],
141              "url_pattern": "/users",
142              "extra_config": {
143                "modifier/martian": {
```

```
144                 "header.Modifier": {
145                 "scope": ["request"],
146                 "name": "Authorization",
147                 "value": "{{ include "/etc/krakend/secrets/basic-
                       auth" }}"
148                 }
149               }
150             }
151           }
152       ]
153     },
154     {
155       "endpoint": "/users/{user}",
156       "input_headers": ["X-Real-IP"],
157       "method": "PATCH",
158       "output_encoding": "json",
159       "extra_config": {
160           "auth/validator": {
161               "alg": "RS256",
162               "audience": ["api://audience"],
163               "jwk_url": "https://dev-5bkfe0he8u5v6zq3.uk.auth0.
                     com/.well-known/jwks.json",
164               "scopes_key": "scope",
165               "scopes_matcher": "any",
166               "scopes": [
167                 "write:users"
168               ],
169               "cache": true
170           },
171           "qos/ratelimit/router": {
172             "max_rate": 5,
173             "client_max_rate": 5,
174             "every": "1m",
175             "strategy": "header",
176             "key": "X-Real-IP"
177         }
178       },
179       "backend": [
180           {
```

```
181             "method": "PATCH",
182             "host": [
183               "http://host.docker.internal:5000"
184             ],
185             "url_pattern": "/users/{user}",
186             "extra_config": {
187               "modifier/martian": {
188               "header.Modifier": {
189               "scope": ["request"],
190               "name": "Authorization",
191               "value": "{{ include "/etc/krakend/secrets/basic-
                     auth" }}"
192               }
193              }
194             }
195           }
196         ]
197       },
198       {
199         "endpoint": "/users/{user}",
200         "input_headers": ["X-Real-IP"],
201         "method": "DELETE",
202         "output_encoding": "json",
203         "extra_config": {
204             "auth/validator": {
205                 "alg": "RS256",
206                 "audience": ["api://audience"],
207                 "jwk_url": "https://dev-5bkfe0he8u5v6zq3.uk.auth0.
                     com/.well-known/jwks.json",
208                 "scopes_key": "scope",
209                 "scopes_matcher": "any",
210                 "scopes": [
211                   "write:users"
212                 ],
213                 "cache": true
214             },
215             "qos/ratelimit/router": {
216               "max_rate": 5,
217               "client_max_rate": 5,
```

```
218                 "every": "1m",
219                 "strategy": "header",
220                 "key": "X-Real-IP"
221             }
222         },
223         "backend": [
224             {
225                 "method": "DELETE",
226                 "host": [
227                   "http://host.docker.internal:5000"
228                 ],
229                 "url_pattern": "/users/{user}",
230                 "extra_config": {
231                   "modifier/martian": {
232                   "header.Modifier": {
233                   "scope": ["request"],
234                   "name": "Authorization",
235                   "value": "{{ include "/etc/krakend/secrets/basic-
                         auth" }}"
236                   }
237                  }
238                 }
239             }
240         ]
241       }
242     ],
243     "extra_config": {
244       "plugin/http-server": {
245         "name": ["global-rate-limit-header"],
246         "global-rate-limit-header": {
247                 "host": "host.docker.internal",
248                 "port": "6379",
249                 "password": "{{ include "/etc/krakend/secrets/redis/
                      password" }}",
250                 "header": "X-Real-IP",
251                 "rate": "10",
252                 "period": "minute"
253         }
254       },
```

```
255     "telemetry/influx": {
256         "address": "http://host.docker.internal:8086",
257         "ttl": "25s",
258         "buffer_size": 0,
259         "db": "krakend_local",
260         "username": "{{ include "/etc/krakend/secrets/influxdb/
              INFLUXDB_USER" }}",
261         "password": "{{ include "/etc/krakend/secrets/influxdb/
              INFLUXDB_PASSWORD" }}"
262     },
263     "telemetry/metrics": {
264       "collection_time": "30s",
265       "listen_address": "127.0.0.1:{{ env "KRAKEND_PORT_METRICS"
            }}"
266     },
267     "telemetry/logging": {
268       "level": "WARNING",
269       "@comment": "Prefix should always be inside [] to keep the
            grok expression working",
270       "prefix": "[KRAKEND]",
271       "syslog": false,
272       "stdout": true
273     },
274     "telemetry/gelf": {
275       "address": "host.docker.internal:12201",
276       "enable_tcp": false
277     },
278     "telemetry/opencensus": {
279       "sample_rate": 100,
280       "reporting_period": 0,
281       "exporters": {
282         "jaeger": {
283           "endpoint": "http://host.docker.internal:14268/api/
                traces",
284           "service_name":"krakend",
285           "buffer_max_count": 1000
286         }
287       }
288     },
```

```
289     "security/http": {
290       "ssl_proxy_headers": {
291         "X-Forwarded-Proto": "https"
292       },
293       "host_proxy_headers": [
294         "X-Forwarded-Hosts",
295         "X-Forwarded-For",
296         "X-Real-IP"
297       ],
298       "ssl_redirect": true,
299       "ssl_host": "host.docker.internal:{{ env "KRAKEND_PORT" }}",
300       "sts_seconds": 31536000,
301       "sts_include_subdomains": true,
302       "frame_deny": true,
303       "custom_frame_options_value": "DENY",
304       "referrer_policy": "no-referrer",
305       "content_type_nosniff": true,
306       "browser_xss_filter": false,
307       "content_security_policy": "default-src 'none';",
308       "ssl_port": "{{ env "KRAKEND_PORT" }}"
309     }
310   }
311 }
```

Listing A.1: KrakenD config to protect sample API

```go
1  // SPDX-License-Identifier: Apache-2.0
2
3  package main
4
5  import (
6      "context"
7      "crypto/tls"
8      "errors"
9      "fmt"
10     "net/http"
11     "strconv"
12     "strings"
13     "sync"
14     "time"
15
16     redis "github.com/redis/go-redis/v9"
```

```
17     redis_rate "github.com/go-redis/redis_rate/v10"
18 )
19
20 const (
21     loggerPrefix           = "[PLUGIN: global-rate-limit-header] "
22     retryConnectionSeconds = 10
23 )
24
25 // pluginName is the plugin name
26 var pluginName = "global-rate-limit-header"
27
28 // HandlerRegisterer is the symbol the plugin loader will try to load.
   It must implement the Registerer interface
29 var HandlerRegisterer = registerer(pluginName)
30
31 type registerer string
32
33 func (r registerer) RegisterHandlers(f func(
34     name string,
35     handler func(context.Context, map[string]interface{}, http.Handler)
       (http.Handler, error),
36 )) {
37     f(string(r), r.registerHandlers)
38 }
39
40 func loopUntilRedisConnectionFound(rdb *redis.Client, ctx context.
   Context, isTryingToReconnectMu *sync.Mutex, isTryingToReconnect *bool
   ) {
41
42     logger.Debug(fmt.Sprintf(loggerPrefix + " testing redis connection
       ..."))
43
44     ticker := time.NewTicker(retryConnectionSeconds * time.Second)
45     defer ticker.Stop()
46
47     for {
48         select {
49         case <-ticker.C:
50             if _, err := rdb.Ping(ctx).Result(); err != nil {
51                 logger.Debug(fmt.Sprintf(loggerPrefix + " redis
                   connection not working, retrying in 10s..."))
52             } else {
53                 isTryingToReconnectMu.Lock()
54                 *isTryingToReconnect = false
```

```
55                  isTryingToReconnectMu.Unlock()
56                  logger.Debug(fmt.Sprintf(loggerPrefix + " redis
                        connection OK!"))
57                  return
58              }
59          }
60      }
61  }
62
63  func (r registerer) registerHandlers(_ context.Context, extra map[string
        ]interface{}, h http.Handler) (http.Handler, error) {
64      // If the plugin requires some configuration, it should be under the
            name of the plugin. E.g.:
65      /*
66              "extra_config":{
67                  "plugin/http-server":{
68                      "name":["global-rate-limit-header"],
69                      "global-rate-limit-header":{
70                          "host": "<REDIS_HOST_NAME>",
71                          "port": "<REDIS_HOST_PORT",
72                          "password": "<REDIS_PASSWORD>",
73                          "header": "<HEADER_NAME>",
74                          "rate": <MAX_RATE>",
75                          "period": "<minute> || <second>""
76                      }
77                  }
78              }
79      */
80      // The config variable contains all the keys you have defined in the
            configuration
81      // if the key doesn't exists or is not a map the plugin returns an
           error and the default handler
82      config, ok := extra[pluginName].(map[string]interface{})
83      if !ok {
84          return h, errors.New("configuration not found")
85      }
86
87      // read here configs from config file
88
89      host, _ := config["host"].(string)
90      logger.Debug(fmt.Sprintf(loggerPrefix+"redis host: %s", host))
91
92      port, _ := config["port"].(string)
93      logger.Debug(fmt.Sprintf(loggerPrefix+"redis port: %s", port))
```

```
94
95      password, _ := config["password"].(string)
96
97      headerName, _ := config["header"].(string)
98      logger.Debug(fmt.Sprintf(loggerPrefix+"header: %s", headerName))
99
100     rate, _ := config["rate"].(string)
101     logger.Debug(fmt.Sprintf(loggerPrefix+"rate: %s", rate))
102
103     rateNum, err := strconv.Atoi(rate)
104     if err == nil {
105         logger.Debug(fmt.Sprintf(loggerPrefix+"rate: %d", rateNum))
106     } else {
107         logger.Error(fmt.Sprintf(loggerPrefix + "error while decoding
                rate from configs"))
108     }
109
110     period, _ := config["period"].(string)
111     logger.Debug(fmt.Sprintf(loggerPrefix+"period: %s", period))
112
113     isPerMinute := true
114     if strings.ToLower(period) == "second" {
115         isPerMinute = false
116     }
117
118     // initialize connection to the redis db here
119
120     rdb := redis.NewClient(&redis.Options{
121         Addr:     host + ":" + port,
122         Password: password,
123         DB:       0, // use default DB
124         TLSConfig: &tls.Config{
125             MinVersion:         tls.VersionTLS12,
126             InsecureSkipVerify: true, //used for self-signed
                    certificates
127         },
128     })
129
130     ctx := context.Background()
131
132     // global boolean value
133     var isTryingToReconnectMu sync.Mutex
134     isTryingToReconnect := false
135
```

```
136     // CHECK CONNECTION TO REDIS
137     loopUntilRedisConnectionFound(rdb, ctx, &isTryingToReconnectMu, &
            isTryingToReconnect)
138 
139     // return the actual handler wrapping or your custom logic so it can
             be used as a replacement for the default http handler
140     return http.HandlerFunc(func(w http.ResponseWriter, req *http.
            Request) {
141 
142         // GLOBAL RATE LIMIT LOGIC
143 
144         if isTryingToReconnect {
145             logger.Debug(loggerPrefix + "global rate limit DISABLED,
                    forward request")
146             h.ServeHTTP(w, req)
147             return
148         }
149 
150         // get specified header from the request
151         header := req.Header.Get(headerName)
152         logger.Debug(loggerPrefix + "Read " + headerName + " : " +
                header)
153 
154         var (
155             res     *redis_rate.Result
156             limiter *redis_rate.Limiter
157             err     error
158         )
159 
160         limiter = redis_rate.NewLimiter(rdb)
161 
162         // rate limite per minute or per second
163         if isPerMinute {
164             res, err = limiter.Allow(ctx, header, redis_rate.PerMinute(
                    rateNum))
165         } else {
166             res, err = limiter.Allow(ctx, header, redis_rate.PerSecond(
                    rateNum))
167         }
168 
169         // recover from lost redis connection
170         defer func() {
171             if r := recover(); r != nil {
172                 logger.Debug(fmt.Sprintf(loggerPrefix + " lost redis
```

```
                connection. trying to recover..."))
173
174             isTryingToReconnectMu.Lock()
175             start := !isTryingToReconnect
176             isTryingToReconnect = true
177             isTryingToReconnectMu.Unlock()
178             if start {
179                 go loopUntilRedisConnectionFound(rdb, ctx, &
                    isTryingToReconnectMu, &isTryingToReconnect)
180             }
181
182             logger.Debug(loggerPrefix + "global rate limit DISABLED,
                    forward request")
183             h.ServeHTTP(w, req)
184             return
185         }
186     }()
187
188     if err != nil {
189         // recover from this panic to handle lost connection
190         panic(err)
191     }
192
193     logger.Debug(fmt.Sprintf(loggerPrefix+"header: %s +1", header))
194     logger.Debug(fmt.Sprintf(loggerPrefix + "allowed " + strconv.
        Itoa(res.Allowed) + " remaining %s " + strconv.Itoa(res.
        Remaining)))
195
196     if res.Allowed == 0 {
197
198         seconds := int(res.RetryAfter / time.Second)
199
200         // generate too many requests response
201         logger.Debug(loggerPrefix + "global rate limit reached, stop
                request")
202         errorMessage := "Too Many Requests: retry after " + strconv.
            Itoa(seconds) + " seconds"
203         http.Error(w, errorMessage, http.StatusTooManyRequests)
204         return
205
206     } else {
207
208         logger.Debug(loggerPrefix + "global rate limit ok, forward
            request")
```

```
209             h.ServeHTTP(w, req)
210             return
211         }
212
213     }), nil
214 }
215
216 func main() {}
217
218 // This logger is replaced by the RegisterLogger method to load the one
    from KrakenD
219 var logger Logger = noopLogger{}
220
221 func (registerer) RegisterLogger(v interface{}) {
222     l, ok := v.(Logger)
223     if !ok {
224         return
225     }
226     logger = l
227     logger.Debug(fmt.Sprintf("[PLUGIN: %s] Logger loaded",
        HandlerRegisterer))
228 }
229
230 type Logger interface {
231     Debug(v ...interface{})
232     Info(v ...interface{})
233     Warning(v ...interface{})
234     Error(v ...interface{})
235     Critical(v ...interface{})
236     Fatal(v ...interface{})
237 }
238
239 // Empty logger implementation
240 type noopLogger struct{}
241
242 func (n noopLogger) Debug(_ ...interface{})    {}
243 func (n noopLogger) Info(_ ...interface{})     {}
244 func (n noopLogger) Warning(_ ...interface{})  {}
245 func (n noopLogger) Error(_ ...interface{})    {}
246 func (n noopLogger) Critical(_ ...interface{}) {}
247 func (n noopLogger) Fatal(_ ...interface{})    {}
```

Listing A.2: KrakenD HTTP server plugin to enable global rate limit with external redis database based on HTTP header, starting from the HTTP server plugin example in the official documentation

```
# BUILDER
FROM golang:1.21rc2-alpine3.18 as builder

ARG KRAKEND_CONFIG_PATH=configs/backend/krakend.tmpl
ENV ENV_KRAKEND_CONFIG_PATH=$KRAKEND_CONFIG_PATH

RUN apk --no-cache --virtual .build-deps add make gcc musl-dev binutils-
    gold

# COPY KRAKEND CODE
COPY ./krakend-ce /app

# COPY PLUGINS CODE
COPY ./plugins /app/plugins

# COPY CONFIGS
COPY ./configs /app/configs

# COPY INTEGRATION TEST
COPY ./tests/krakend-integration /app/tests/krakend-integration

# BUILD PLUGINS

ENV PLUGINS_LIST="\
    /app/plugins/basic-auth/basic-auth-global \
    /app/plugins/basic-auth/basic-auth-partial \
    /app/plugins/rate-limit/global-rate-limit-header \
    /app/plugins/rate-limit/global-rate-limit-ip \
    /app/plugins/rate-limit/global-rate-limit \
    /app/plugins/revoker-helper \
"

RUN mkdir -p /app/plugins/compiled/

RUN for plugin in $PLUGINS_LIST; do \
    cd $plugin && \
    go mod tidy -compat=1.17 && \
    go build -buildmode=plugin -o plugin.so . && \
    plugin_name=$(basename $plugin) && \
    mv plugin.so /app/plugins/compiled/$plugin_name.so; \
done


WORKDIR /app
```

```
44
45 # BUILD KRAKEND
46 RUN make build
47
48 # CHECK SYNTAX
49 ENV FC_ENABLE 1
50 RUN ./krakend check -t -d -c ./${ENV_KRAKEND_CONFIG_PATH}
51
52 # AUDIT
53 RUN ./krakend audit -s CRITICAL -c ./${ENV_KRAKEND_CONFIG_PATH}
54
55 # INTEGRATION TEST TOOL
56 WORKDIR /app/cmd/krakend-integration
57 RUN go mod tidy -compat=1.17
58 RUN go run main.go -krakend_bin_path ./../../krakend \
59 -krakend_config_path /app/${ENV_KRAKEND_CONFIG_PATH} \
60 -krakend_specs_path /app/tests/krakend-integration
61
62 # RUNNER
63
64 FROM alpine:3.18
65
66 LABEL maintainer="community@krakend.io"
67
68 RUN apk add --no-cache --repository http://dl-cdn.alpinelinux.org/alpine
      /v3.18/main ca-certificates curl
69
70 RUN apk add --no-cache ca-certificates && \
71     adduser -u 1000 -S -D -H krakend && \
72     mkdir /etc/krakend
73
74 COPY --from=builder /app/krakend /usr/bin/krakend
75
76 ARG KRAKEND_CONFIG_PATH=configs/backend/krakend.tmpl
77 ENV ENV_KRAKEND_CONFIG_PATH=$KRAKEND_CONFIG_PATH
78
79 # COPY CONFIGURATIONS
80 COPY --from=builder /app/${ENV_KRAKEND_CONFIG_PATH} /etc/krakend/krakend
      .tmpl
81
82 # copy compiled plugin into RUNNER
83 COPY --from=builder /app/plugins/compiled/ /usr/bin/plugins/
84
85 # copy public/private tls cert into RUNNER
```

```
86 COPY --from=builder /app/configs/backend/tls /etc/krakend/tls
87
88 USER 1000
89
90 # enable krakend template config file loading
91 ENV FC_ENABLE 1
92
93 ENTRYPOINT [ "/usr/bin/krakend" ]
94
95 WORKDIR /etc/krakend
96
97 CMD [ "run", "-c", "/etc/krakend/krakend.tmpl" ]
98
99 EXPOSE 8080 8090
```

Listing A.3: Dockerfile that generates an immutable docker image containing KrakenD along with the specified plugins, the TLS certificates and the configuration file embedded in it, specified as build argument, while doing build tests. Modified from the original Dockerfile contained in KrakenD.

List of Figures

2.1 KrakenD's plugins types, Extracted from [30] 10

3.1 Scenario of our analysis 14
3.2 Where CEL evaluation can take place, Extracted from [17] 16

4.1 Initial step 27
4.2 Step 1 28
4.3 Step 2 28
4.4 Step 3 29
4.5 Step 4 - Rate limiting scenarios 30
4.6 Step 4 30
4.7 Step 5 31

5.1 High availability cluster 38
5.2 Load balancer architecture with TLS from client and mutual TLS with KrakenD instances 41
5.3 Centralized revoke server architecture, Extracted from [29] 45
5.4 Rate limiting on clusters 46
5.5 Jenkins CI/CD pipeline structure 55
5.6 Cluster recommended network architecture 56
5.7 Gatling simulation before breaking point 59
5.8 Gatling simulation after breaking point 59

Listings

2.1 KrakenD's configuration file main structure, extracted from [14] 7
2.2 KrakenD's endpoints configuration sample 8
2.3 KrakenD's extra_config structure, extracted from [14] 9
3.1 KrakenD's CEL validation rule example to check that the "sub" claim inside the payload correspond to the {user} parameter of the request . . . 16
3.2 KrakenD's sample endpoint configuration to filter response using the allow list feature, filtering the response to return only the "name" field 19
3.3 KrakenD's JWT role validation endpoint example 21
5.1 KrakenD `/users` endpoint configuration to let the API gateway act as a proxy 34
5.2 Curl request to get an access token from Auth0 using the client credentials flow 36
5.3 KrakenD configuration to enable JWT validation on an endpoint 37
5.4 Load balancer basic configuration to split traffic amongst KrakenD instances using a round robin balancing algorithm 39
5.5 Nginx configuration to split traffic amongst KrakenD instances using a round robin balancing algorithm, with TLS from client to load balancer and mTLS between load balancer and KrakenD instances 40
5.6 KrakenD sample config to enable mutualTLS 41
5.7 KrakenD example configuration to enable both absolute rate limiting and client rate limiting on an endpoint, identified by different X-Real-IP header 47
5.8 KrakenD example configuration for stateless rate limiting on clusters, splitting the traffic of the precedent configuration across two instances 48
5.9 Global rate limit plugin logic 49
5.10 KrakenD example configuration to set up stateful global rate limit plugin, identifying client by the X-Real-IP HTTP header, with a rate limit of 30 requests per minute, to be placed as extra_config at root level 50
5.11 KrakenD config to export logs to logstash as extra_config to be placed at root level, extracted from [18] 52

5.12 KrakenD example configuration to export metrics to influxdb, as extra_config to be placed at root level, extracted from [20] 52
5.13 KrakenD example configuration to export traces to Jaeger as extra_config to be placed at root level, extracted from [21] 53
5.14 KrakenD example configuration using Go templates to dinamically inject redis password into the configuration from a file 54
5.15 Hey local tests commands 57
A.1 KrakenD config to protect sample API 67
A.2 KrakenD HTTP server plugin to enable global rate limit with external redis database based on HTTP header, starting from the HTTP server plugin example in the official documentation 75
A.3 Dockerfile that generates an immutable docker image containing KrakenD along with the specified plugins, the TLS certificates and the configuration file embedded in it, specified as build argument, while doing build tests. Modified from the original Dockerfile contained in KrakenD. 82

List of Tables

3.1 Qualitative scale definition 25
3.2 Evaluation analysis of the role of API gateways in mitigating most critical API security risks 26

5.1 Sample API definition 33
5.2 Centralized revoker strategy recap 44
5.3 Comparison of different JWT revocation methods, extracted from [13] and modified 45
5.4 Revoke server endpoints 45
5.5 Comparison of stateless and stateful rate limiting approaches 50
5.6 Sample firewall rules for stateful packet filter 56
5.7 Local tests with Hey 57
5.8 Local tests with Hey with load balancer 57
5.9 Gatling simulation endpoints requests for each simulated user 59

Glossary

API Application Programming Interface. i, iii, 1, 2, 5, 9, 13, 14, 17, 18, 22, 58, 62

BOLA Broken Object Level Authorization. 15

CI/CD Continous Integration / Continous Delivery. 22, 37, 54

CRUD Create Read Update Delete. 33

GW Gateway. 58

HTTP Hypertext Transfer Protocol. 24, 29, 47–49, 56, 58

IdP Identity Provider. 2, 13, 17, 28, 32, 35, 42

IP Internet Protocol. 48

JSON Javascript Object Notation. 7, 37

JWT Json Web Token. 15, 18, 35, 36

OIDC OpenID Connect. 13, 17, 35

OWASP Open Web Application Security Project. 14, 15

PoC Proof of Concept. 2

REST Representational State Transfer. 33

RPC Remote Procedure Call. 39

SaaS Software as a Service. 35

SSRF Server Side Request Forgery. 24

TLS Transport Layer Security. 5