How to automatically validate the configuration of your API Gateway?

September 20, 2021

Share

API everywhere…

Today, it is common for software, companies, etc. to provide a web API to expose data to their customers or partners[1]. The objective is to facilitate the integration between Information Systems and create new business opportunities. For example, for banks, API was a way to provide more services to their customers through mobile applications. Do you remember the last time you needed to contact your bank directly or go physically to your bank agency?

The era of API gateways

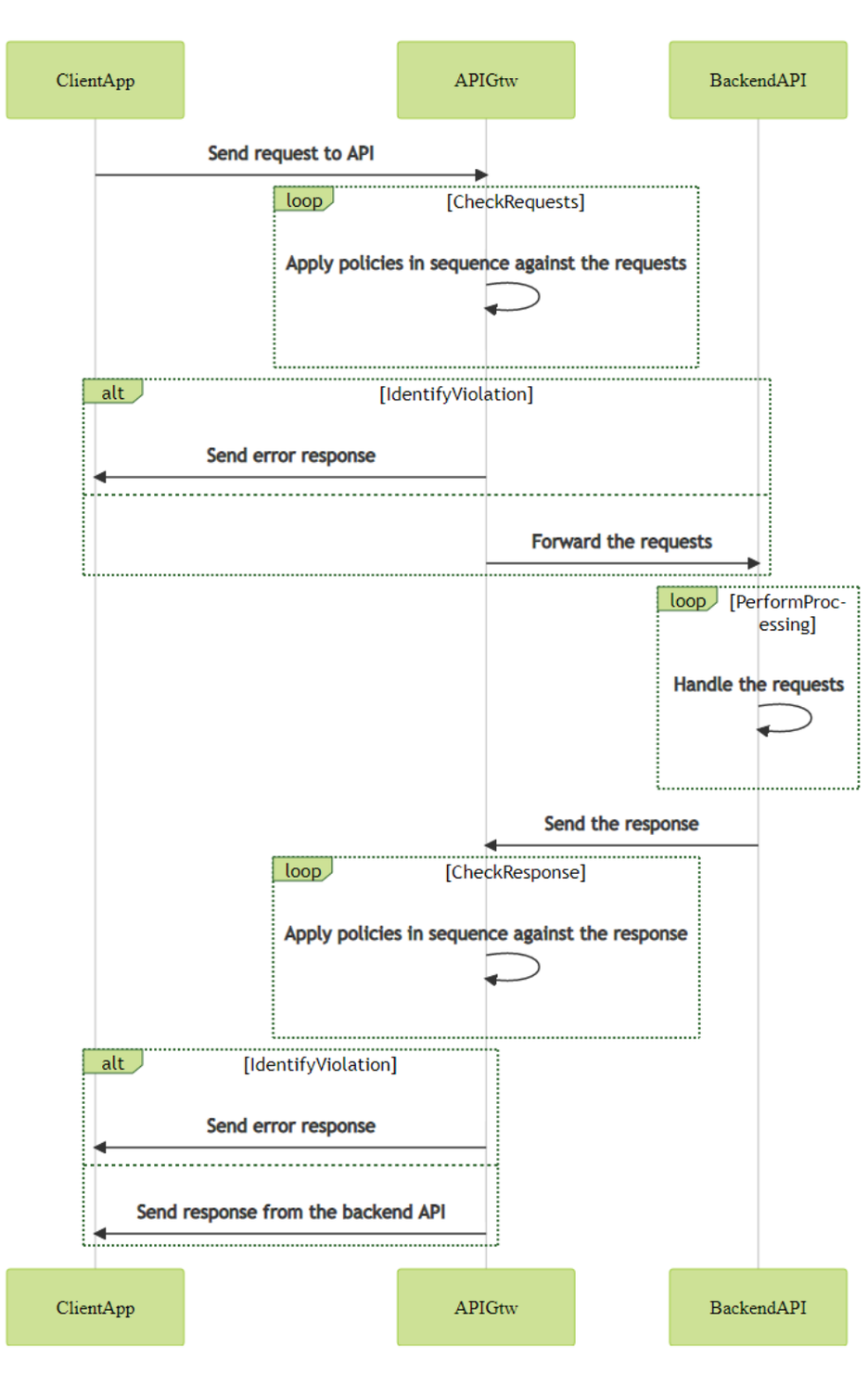
With the increase of API created, companies needed to find a way to "*easily*" manage different aspects of the API like exposure, monetization, access control, documentation, versioning, aggregate services, etc.

Therefore, API Gateway[2] was born to achieve these goals. Even if the objective of this post is not to describe "*what an API Gateway is?*" but rather "*How to test its configuration?*" Let' see what is its role in an API context.

The goal of an API Gateway is to be a central access point for any call to an API offered to client applications. Therefore, to be effective, all calls to API must be routed, without exception. In this way, Backend API (internal API performing the real business processing) can delegate several aspects to the API Gateway like authentication, authorization, rate limiting, modifies the requests/responses, etc.

Overview of the communication flow

The schema below shows an overview of the flow involved during the call to an API from a client application. A "*policy*", refers to a set of processing/validation rules applied on a request or a response.



You should not (by) pass the API Gateway!

When an API Gateway is deployed to handle access to a backend API, it is important to ensure that only the API Gateway is allowed to call the backend API. The following kind of alternate path must be avoided:

The path in the dotted line represents a call channel that bypasses all the protection and restriction applied by the API Gateway. To restrict backend API calls to the API Gateway, the following measure, among others, can be leveraged:

- Mutual TLS authentication between the API Gateway and the Backend API.
- Network segregation alongside specific firewall rules.

Depending on the API Gateway software, different kinds of built-in measures are supported out of the box.

The challenge

Even if an API Gateway helps manage the exposure of API, with time, it will contain a significant amount of API definition including different versions of the same API. The more API definition an API Gateway contains, the more is difficult to ensure that each configuration stays secure across the different iteration of the API Gateway configuration.

The common pattern meet is the "*Configure, test and forget*". Precisely, an API is configured in the API Gateway, the configuration is validated by a configuration or an intrusion test against the API and then the API lives its life until someone notices a problem or an incident happen…

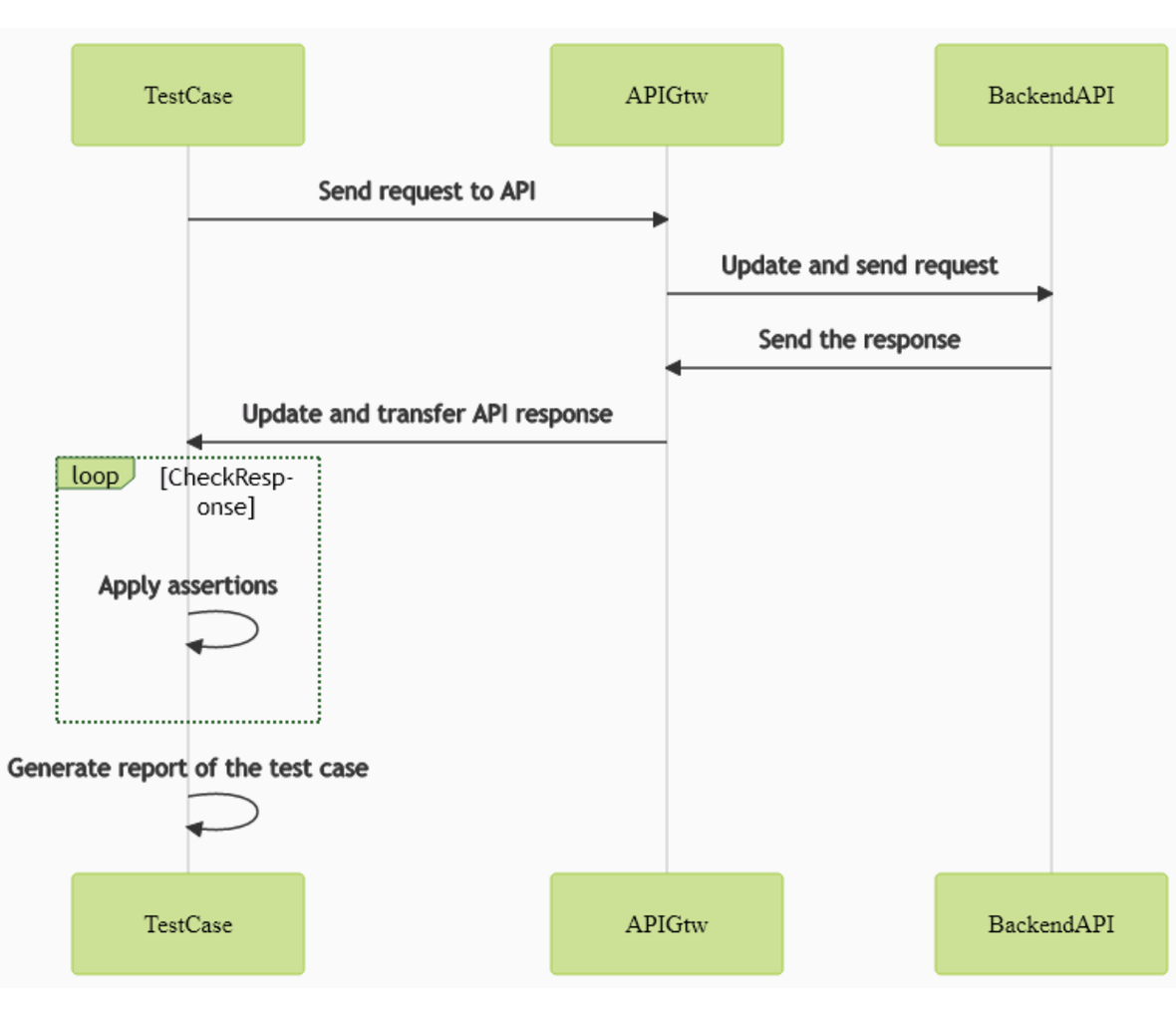
Automation to rule them all

In software development, a kind of test, named "Integration testing"[3], is performed "*to evaluate the compliance of a system or component with specified functional requirements.*"

The idea will be to apply the same kind of test on an API definition, in an automated way, in order to constantly ensure that it stays secure. To fulfill this objective, it will be great if mainly:

1. The test can be described, without needed to code something, like a cooking recipe.
2. The test recipe can define assertions on results.
3. The test syntax can be easy to read and understand.
4. The test can generate reports that can be integrated into popular CI/CD platforms[4].
5. The test description file support versioning.
6. The test tool can be cross-platform and require no or minimal installation.
7. The test tool can be free, open-source, maintained and well documented.

The test flow for an API definition will be the following:



For this purpose, the tool, named Venom[5], was chosen because it fulfills most of the criteria mentioned above.

Below is an example of a test case description (more on this later) – No code needed:

```
# The test case is named Strict-Transport-Security because
# it checks the presence of this security HTTP response header
– name: Strict-Transport-Security
  steps:
# We want to perform an HTTP request
  – type: http
# The request is a GET
    method: GET
# To the URL referenced by the variable target_site
    url: {{.target_site}}
# As we will only check the headers then body is not
# important and can be skipped
    skip_body: true
# We allow 20 seconds to the requests to finish
    timeout: 20
# We define a list of assertions
    assertions:
# HTTP response code must be HTTP 200 OK
    – result.statuscode ShouldEqual 200
# The header Strict-Transport-Security must be present with a non-empty value
    – result.headers.strict-transport-security ShouldNotBeNil
# The header Strict-Transport-Security must have a value containing the string specified
    – result.headers.strict-transport-security ShouldContainSubstring "includeSubDomains"
```

The global lab

To confirm that the idea was viable or not, a lab, based on the following technical components, was created:

- API Gateway: **Apiman** from Red Hat (free and open-source)[6].
- Test tool: **Venom** from OVH (free and open source)[7].
- Provisioning & infrastructure: **Docker** and **Docker Compose** (free and open-source)[8].

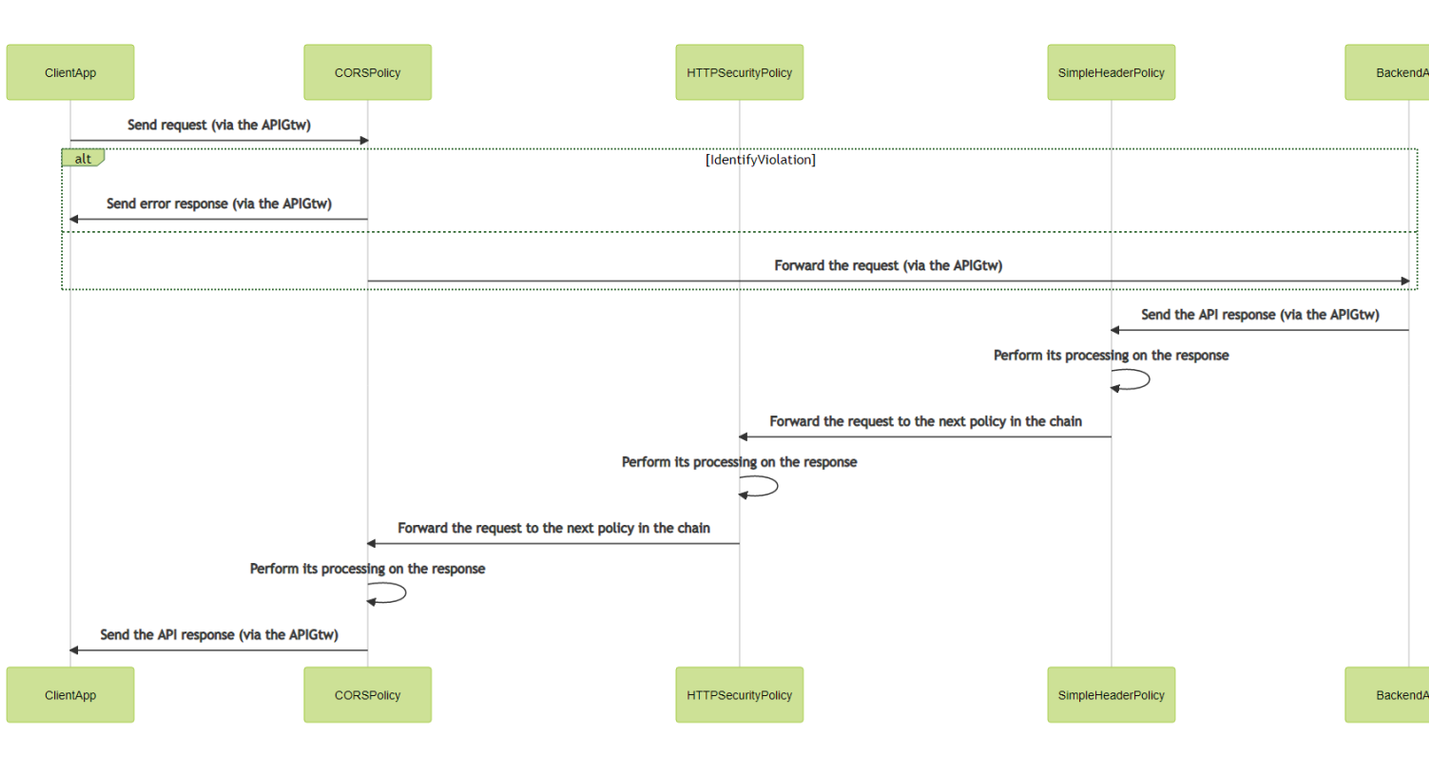
The lab definitions

Two APIs were defined in the API Gateway used for the lab:

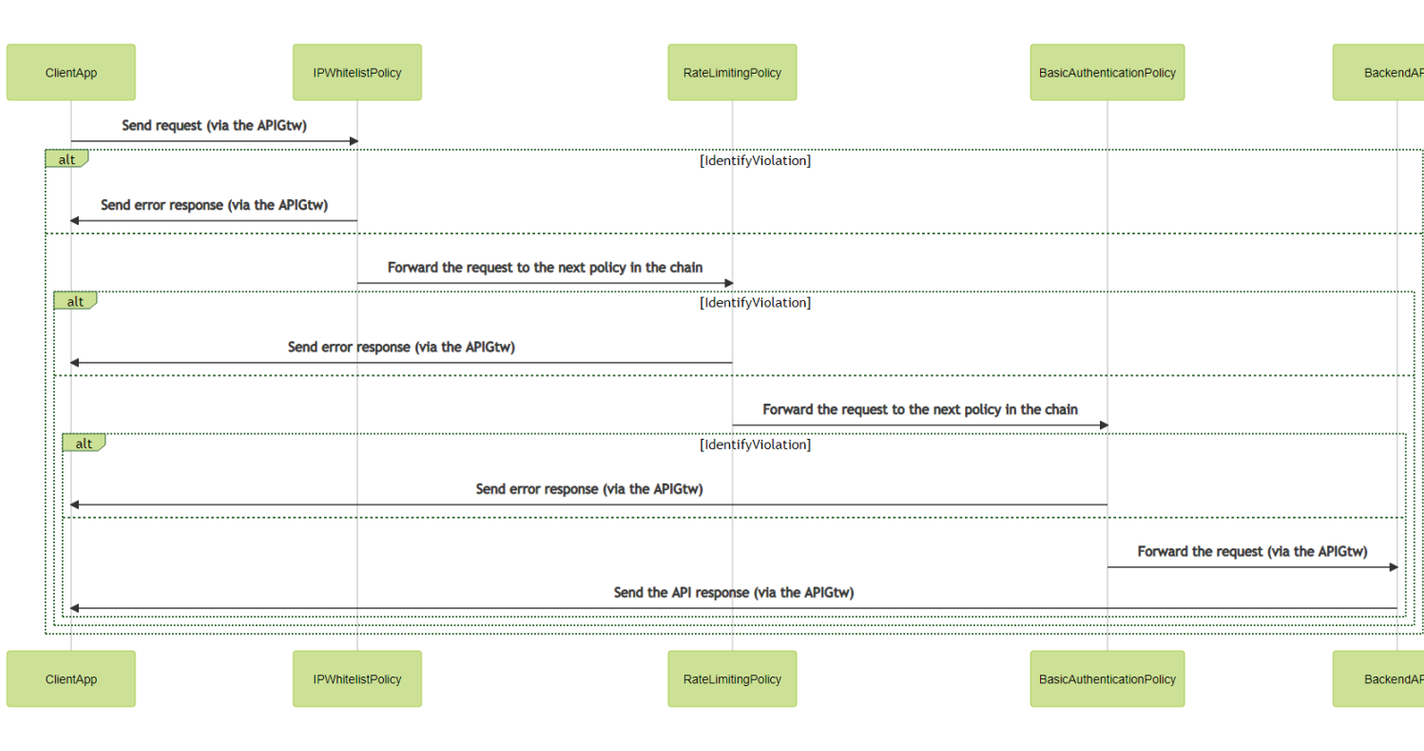
1. One using the API from https://requestbin.net/, as backend API – named **public**:
   - RequestBin gives you a URL that will collect requests made to it and let you inspect them in a human-friendly way.
   - This API is public and it is not published through a "*Plan*".
   - Policies applied via built-in plugins[9]:
     - **CORS**: Only allow origins https://localhost:8443, http://localhost:8080 and apply cache of 10 seconds.[10]
     - **HTTP Security**: Enable **Content-Security-Policy** / **X-Frame-Options** / **X-Content-Type-Options** headers. Disable **HSTS** (local POC) / **X-XSS-Protection** headers.[11]
     - **Simple Header**: Remove the following headers from backend API response: **cf-request-id**, **Report-To**, **Server**, **CF-RAY**, **NEL**, **CF-Cache-Status**.
2. One using the API from http://jsonplaceholder.typicode.com/, as backend API – named **published**:
   - Free fake API for testing and prototyping.
   - This API is not public and it is published through a "*Plan*".
   - Policies applied via built-in plugins:
     - **IP Whitelist**: Only allow requests from **127.*.*.*** and **172.*.*.***
     - **Rate Limiting**: Only allow 10 requests by second from the client app.
     - **Basic Authentication**: Require basic authentication from users defined statically and forward the login to the backend API in header **X-User**.

The following schema represents the communication flow.

API named **public**:



API named **published**:



Explanation about the notion of "Plan" in apiman – Extract of the documentation[12]:

"*In apiman, a Plan is a set of policies that together define the level of service that apiman provides for an API. Plans enable apiman users to define multiple different levels of service for their APIs.*"

Explanation of the difference between a "Public API" and "Plan-based" API[13]:

"*Public APIs are also very flexible in that they can be updated without being re-published. Unlike APIs published through Plans,* ***Public API can be accessed by a client app without requiring API consumers to agree to any terms and conditions related to a contract defined in a plan for the API…***"

"*Publishing an API through Plans –* ***In contrast to Public APIs, these APIs, once published, must be accessed by a Client App via its API key.*** *In order to gain access to an API, the Client App must create a contract with an API through one of the API's configured Plans…*"

The lab API test plans

One **Venom** test plan (one test plan contains one or several test cases) by API was created with the objective to ensure that a maximum possible policy in place is covered by, at least, one test case.

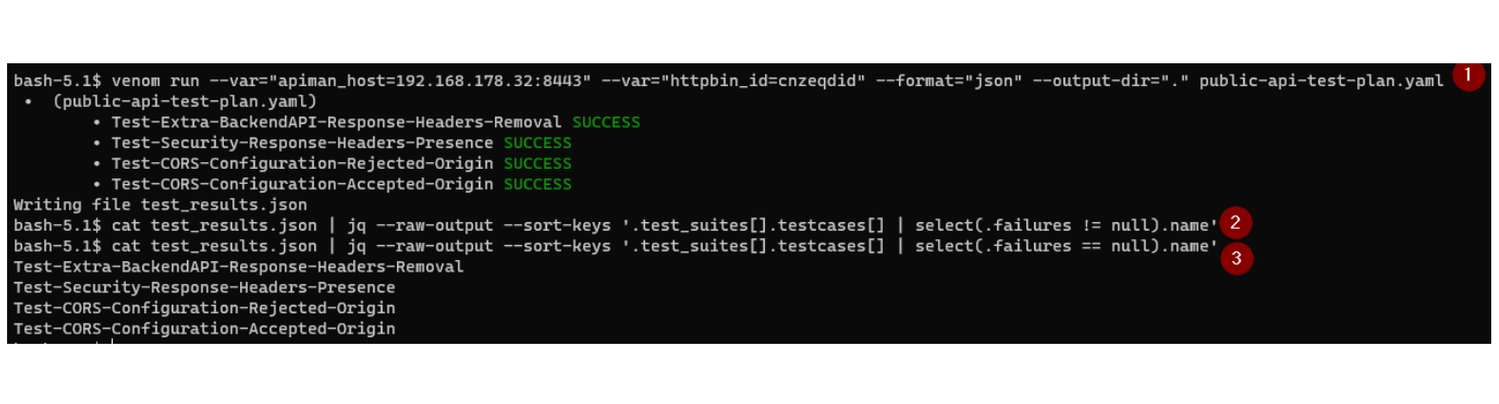
Test plan for the public API – File **public-api-test-plan.yaml**[14] :

Test Case	Objective
Test-Extra-BackendAPI-Response-Headers-Removal	Validate that the policy defined with the plugin "Simple Header" removes all expected headers from the backend API.
Test-Security-Response-Headers-Presence	Validate that the policy defined with the plugin "HTTP Security": 1. Enable expected HTTP security response headers. 2. Disable expected HTTP security response headers.
Test-CORS-Configuration-Rejected-Origin	Validate that the policy defined with the plugin "CORS" rejects the request if the non-allowed value is specified in the "Origin" request header.
Test-CORS-Configuration-Accepted-Origin	Validate that the policy defined with the plugin "CORS" accepts the request if the allowed value is specified in the "Origin" request header.

Test plan for the published API – File **published-api-test-plan.yaml**[15]:

Test Case	Objective
Test-Missing-API-Key	Validate that the API Gateway rejects the request if no API Key is provided.
Test-Non-Verbose-Error	Validate that the API gateway does not return verbose errors in case of policy violation.
Test-Missing-Basic-Authentication	Validate that the policy defined with the plugin "Basic Authentication" rejects the request if invalid credentials are provided.
Test-Valid-Basic-Authentication	Validate that the policy defined with the plugin "Basic Authentication" accepts the request if valid credentials are provided.
Test-Rate-Limiting-Effectiveness	Validate that the policy defined with the plugin "Rate Limiting" blocks access to the API if too many requests are made in the defined period.

Example of execution of the test plan for the public API:



Steps:

1. Execute the test plan and ask **Venom** to generate a JSON file with the execution state results.
2. Extract via **Jq**[16], the list of test cases name that has failed.
3. Extract via **Jq**, the list of test cases name that has succeeded.

The lab API test plans reporting

In addition to the JSON report, it is also possible to generate an output file with the Junit[17] format. This format is supported by default by many CI/CD platform reporting systems.

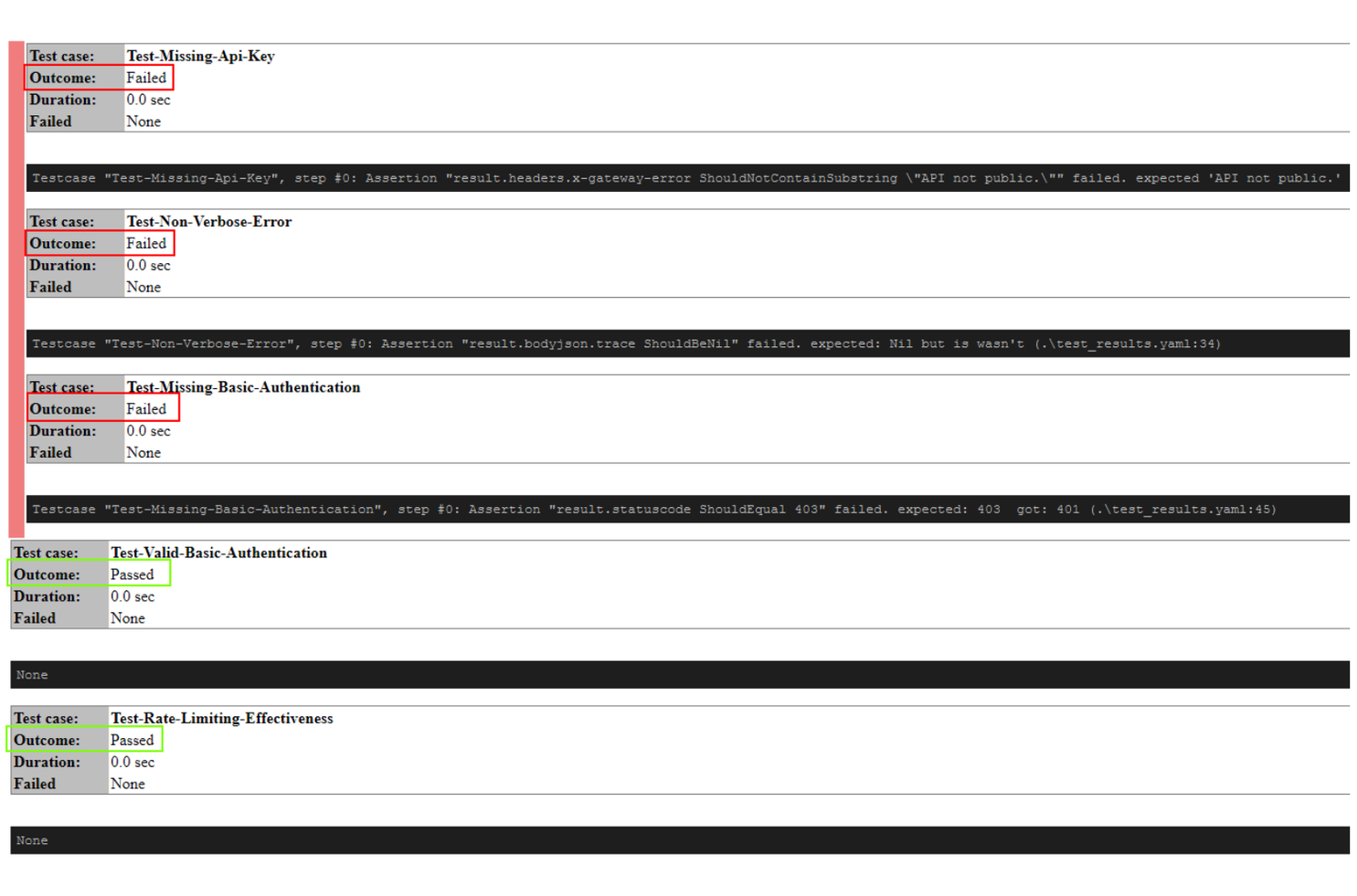
Use the following Venom command, a file named "*test_results.xml*" will be created in the output directory specified:

```
venom run –var="apiman_host=192.168.178.32:8443" –var="httpbin_id=cnzeqdid"
–format="xml" –output-dir="." public-api-test-plan.yaml
```

The following tool[18] can be used to generate an HTML report from the Junit XML report:

```
junit2html test_results.xml test_results.html
```

Overview of the generated HTML report:



The lab playground

The lab, created for this post, provides you with a playground to discover and experiment with different features of Venom in terms of API testing purposes.

Simply follow the instruction defined in the "*Execute the lab*"[19] of the associated GitHub repository and explore the test case possibilities…

Conclusion

APIs are now inevitable and will continue to become a more and more important aspect of the exposure of your Information System to your partner/customers.

This post has presented a simple approach, using free and open-source software, in order to automate the security testing of the API definition in place in your API Gateway in the more API Gateway software agnostic possible.

Feel free to use this simple proposal to build your proper API security testing strategy 😊.

PS: If you would like to meet the AppSec team, join us this 7th of October at our annual event Les Rencontres de la Sécurité. We would love to meet you and discuss API Gateway (or other) with you!

Les Rencontres de la Sécurité 7 Billets, Le jeu 7 oct. 2021 à 09:00 | Eventbrite

Credits:

Dominique Righetto.

Advice

Recent Posts

The Necessity of Cyber Crisis Exercises

The Art of Password Spraying: A Comprehensive Analysis

Common client-side vulnerabilities of web applications

Categories

Advice
Consulting
Cyber Blog Post
General
Newsletter
PR
The Cyber Blog Times
Uncategorized

Search

Search …