🔥

API Firewall guide

Introduction

In our modern world, web applications are becoming ever more important. Bad actors know this and they target them more frequently than ever before. This is not likely to stop any time soon as the number of web applications the world needs will only go up with its reliance on technology. To fully prevent an attack is impossible but we need to try our hardest to do so and in our daily struggles in this field we have a weapon that's known as "hardening".

What is hardening

In the most basic sense of the word, hardening your server means to increase its defences but in practice there are so many ways in which a server can be vulnerable. One technique that is commonly used is to sanitise the user input in our application. User data might contain malicious code or unexpected input and we should take care in hardening our servers against this behaviour.

Another hardening technique that is being used quite often is what's called javascript obfuscation. Web applications are no longer the static websites they used to be. They contain a lot of javascript and this javascript could reveal a lot of information to a potential hacker. To prevent this, the code is obfuscated so that the hacker will have a much harder time figuring out the applications internal workings.

Firewalls on the packet level have been implemented for ages in production environments but recently Web Application Firewall's are gaining ground fast. They will inspect any http request and depending on the configuration, they might report or block any call that contains unexpected or malicious input. This is just one tools in the tool belt of the modern system administrator but it's a very important one.

Hardening vs not hardening

Hardening your web applications comes with a cost as do all good things in life. A more hardened server will be more secure from attackers, but it will also be slower as all of these security measures come with a cost. They will require more processing time of the system as a whole and of its components and this might significantly slow down an application.

For this exact reason it's often a balance between implementing too little hardening and implementing too much.

Meet API firewall

API Firewall, is a light-weighted API Firewall to protect your API endpoints in cloud-native environments with API Schema validation. API Firewall relies on a positive security model allowing calls that match predefined API specs, while rejecting everything else.

Technically, API Firewall is a reverse proxy with a built-in OpenAPI v3 request and response validator, written in Go, and optimised for extreme performance and near-zero added latency.

Pre-requisites

To install the API Firewall we have a couple of pre-requisites that need to be satisfied.

- Docker daemon installed (https://www.docker.com/)
- For this guide we will be assuming you are working on a unix based system

- Git needs to be installed (https://www.atlassian.com/git/tutorials/install-git)
- API Firewall, which we will download later on in the guide (https://hub.docker.com/r/wallarm/api-firewall)
- Zalando's connexion demo API repo https://github.com/zalando/connexion

Run these commands first

- apt install docker.io
- apt install python3-pip

Installing the connection demo

To test our WAF we will using the connexion demo provided by Zalando. We need to install and run this of course. We can do this with one easy commands.

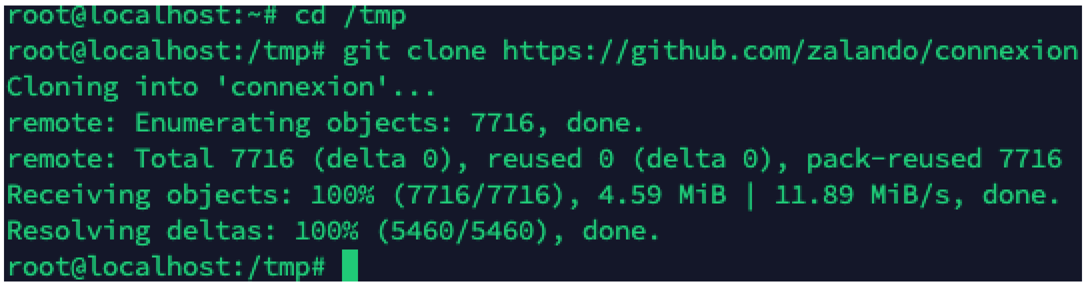
```
pip install connexion[swagger-ui]
```

Later on we will start up connexion so we can test it with our API firewall.

Setting up

To get started we move into the tmp directory before we clone the connexion repository.

```
cd /tmp
git clone https://github.com/zalando/connexion
```



```
root@localhost:~# cd /tmp
root@localhost:/tmp# git clone https://github.com/zalando/connexion
Cloning into 'connexion'...
remote: Enumerating objects: 7716, done.
remote: Total 7716 (delta 0), reused 0 (delta 0), pack-reused 7716
Receiving objects: 100% (7716/7716), 4.59 MiB | 11.89 MiB/s, done.
Resolving deltas: 100% (5460/5460), done.
root@localhost:/tmp#
```

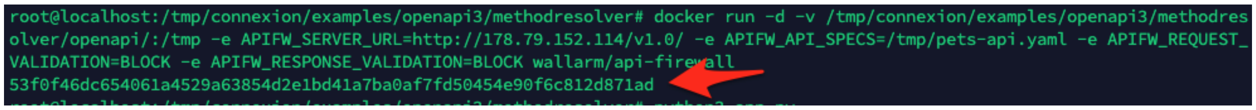
Next we need to pull the api-firewall docker file from the docker hub

```
docker pull wallarm/api-firewall
```

As a last step we can easily run the API firewall with the following command

```
docker run -d -v /tmp/connexion/examples/openapi3/methodresolver/openapi/:/tmp -e APIFW_SE
RVER_URL=http://178.79.152.114:9090/v1.0/ -e APIFW_API_SPECS=/tmp/pets-api.yaml -e APIFW_R
EQUEST_VALIDATION=BLOCK -e APIFW_RESPONSE_VALIDATION=BLOCK  -p 8282:8282 wallarm/api-firew
all
```

This will start up our docker container in the background and give us the identifier of the docker container.



```
root@localhost:/tmp/connexion/examples/openapi3/methodresolver# docker run -d -v /tmp/connexion/examples/openapi3/methodres
olver/openapi/:/tmp -e APIFW_SERVER_URL=http://178.79.152.114/v1.0/ -e APIFW_API_SPECS=/tmp/pets-api.yaml -e APIFW_REQUEST_
VALIDATION=BLOCK -e APIFW_RESPONSE_VALIDATION=BLOCK wallarm/api-firewall
53f0f46dc654061a4529a63854d2e1bd41a7ba0af7fd50454e90f6c812d871ad
```

Starting up the connexion python app is up next because we need to have something to test on. We can do this by moving into the directory "methodresolver" and using python3 to start the app.

```
cd /tmp/connexion-master/examples/openapi3/methodresolver/
python3 app.py
```

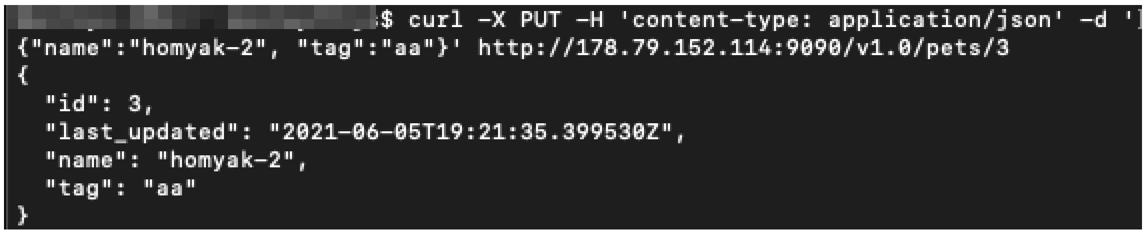
You can check out the URL that it took up in the logs.

Now we need to test that everything works using curl. We will try to make a PUT request to update some data and see if it works. We have to execute this request from an external computer because we will be talking to the public facing IP address of the server. Make sure you replace our IP address with yours.

```
curl -X PUT -H 'content-type: application/json' -d '{"name":"homyak-2", "tag":"aa"}' htt
p://178.79.152.114:8282/v1.0/pets/3
```

This should return the following output.

```
{
  "id": 3,
  "last_updated": "2021-06-05T19:21:35.399530Z",
  "name": "homyak-2",
  "tag": "aa"
}
```



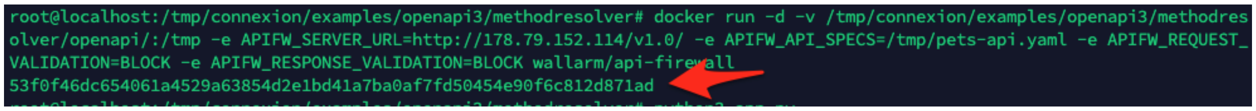
```
[redacted]$ curl -X PUT -H 'content-type: application/json' -d '
{"name":"homyak-2", "tag":"aa"}' http://178.79.152.114:9090/v1.0/pets/3
{
  "id": 3,
  "last_updated": "2021-06-05T19:21:35.399530Z",
  "name": "homyak-2",
  "tag": "aa"
}
```

Now that we have a working API framework and a working API firewall, we can try to send out a malicious request.

```
curl -X PUT -H 'content-type: application/json' -d '{"name":111, "tag":"aa"}' http://178.7
9.152.114:8282/v1.0/pets/3
```

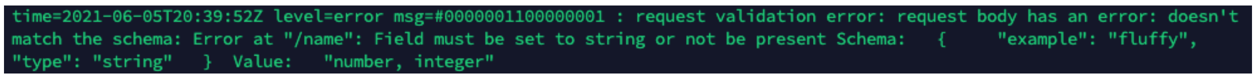
This will return a 403 error response.

To check this on the server we can go back and check the logs of our docker container with the command "docker logs CONTAINERID" where the containerID is the one we got from running our "docker run" command.



```
root@localhost:/tmp/connexion/examples/openapi3/methodresolver# docker run -d -v /tmp/connexion/examples/openapi3/methodres
olver/openapi/:/tmp -e APIFW_SERVER_URL=http://178.79.152.114/v1.0/ -e APIFW_API_SPECS=/tmp/pets-api.yaml -e APIFW_REQUEST_
VALIDATION=BLOCK -e APIFW_RESPONSE_VALIDATION=BLOCK wallarm/api-firewall
53f0f46dc654061a4529a63854d2e1bd41a7ba0af7fd50454e90f6c812d871ad
```

If you lost this ID, that is not a problem, you can check the running docker containers with the "docker container ls". Next we can run the "docker logs" command to view our logs and see that the API firewall is filtering out the requests and blocking them.



```
time=2021-06-05T20:39:52Z level=error msg=#0000001100000001 : request validation error: request body has an error: doesn't
match the schema: Error at "/name": Field must be set to string or not be present Schema:   {     "example": "fluffy",
"type": "string"   }  Value:   "number, integer"
```

The nitty-gritty details

Let's talk a little bit about what we are doing here. First of all we are downloading connexion which is a demo application that will help us test our API firewall. We need to install it via python's pip in order to run it as well.

After pulling our docker container we are starting up our firewalls on port 8282 using our docker container. The following command was used.

```
docker run -d -v /tmp/connexion/examples/openapi3/methodresolver/openapi/:/tmp -e APIFW_SE
RVER_URL=http://178.79.152.114:9090/v1.0/ -e APIFW_API_SPECS=/tmp/pets-api.yaml -e APIFW_R
EQUEST_VALIDATION=BLOCK -e APIFW_RESPONSE_VALIDATION=BLOCK  -p 8282:8282 wallarm/api-firew
all
```

In it we use the following flag to indicate port 8282 should be forwarded.

-p 8282:8282

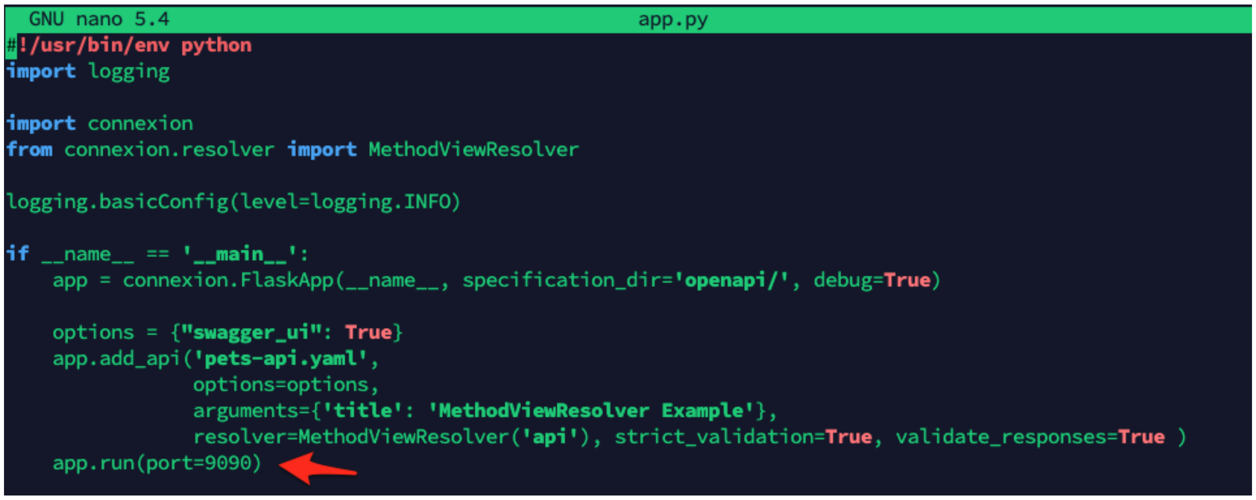
We also use the flag -e APIFW_SERVER_URL=http://178.79.152.114:9090/v1.0/ to indicate where our webserver is running at. This webserver will be made up of the pets example from the connexion demo by later navigating to "/tmp/connexion-master/examples/openapi3/methodresolver/" and starting the app with "python3 app.py". Of course you need to replace the IP adress with the ip address of your own server later on and don't forget to change the port. You can either host your webserver on the same server as the API-firewall or on a different server, in our case we will be running on the same server.

The "-e APIFW_REQUEST_VALIDATION=BLOCK -e APIFW_RESPONSE_VALIDATION=BLOCK " flags ensure that requests that do not pass the validation rules get blocked.

The -v parameter mounts the "/tmp/connexion/examples/openapi3/methodresolver/openapi/" folder on our local drives to the "/tmp" folder on the docker container.

Finally the -d flag ensures that the docker container will run in the background.

No that we are protecting our application, we need to start it as well. When we go to /tmp/connexion/examples/openapi3/methodresolver/ and run "python3 app.py", that's exactly what we. This starts a webserver on port 9090 and serves the API.



```
  GNU nano 5.4                                  app.py
#!/usr/bin/env python
import logging

import connexion
from connexion.resolver import MethodViewResolver

logging.basicConfig(level=logging.INFO)

if __name__ == '__main__':
    app = connexion.FlaskApp(__name__, specification_dir='openapi/', debug=True)

    options = {"swagger_ui": True}
    app.add_api('pets-api.yaml',
                options=options,
                arguments={'title': 'MethodViewResolver Example'},
                resolver=MethodViewResolver('api'), strict_validation=True, validate_responses=True )
    app.run(port=9090)
```

Now we have a working system of an API firewall that filters the traffic going to our API. In a real life scenario, we would now disable access to port 9090 from an outside network and only allow traffic coming from the internal network.