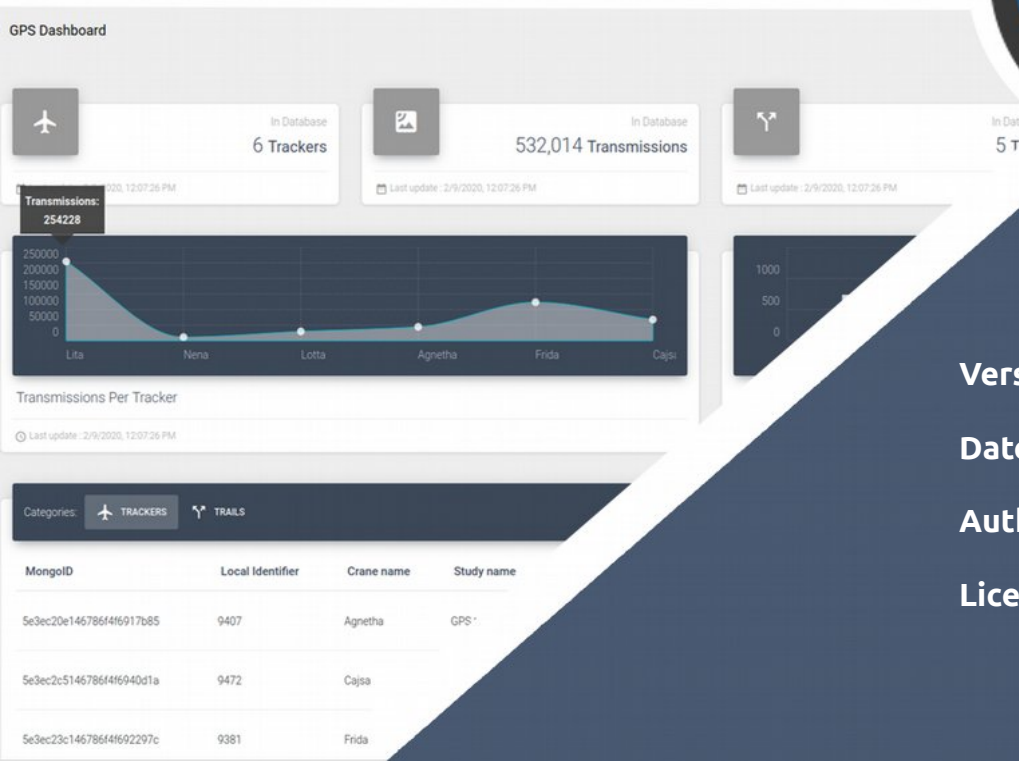
Cookbook

A SECURE NGINX WEB SERVER WITH MODSECURITY



Version : **1.0**

Date : **02-10-2020**

Author : **The GeoStack Project**

Open Content License

Purpose of this document

This cookbook serves as an extension for the following documents:

1) **Cookbook: Creating the GeoStack Course VM:**

   The datastores and tools which we will be using during this cookbook are set up during the cookbook: "Creating the Geostack Course VM"

2) **Cookbook: ETL-Process with datasets:**

   In the cookbook: "ETL-Process with datasets", we learned how to analyze the dataset and filter the data that serves the needs of our application.

3) **Cookbook: Data Modeling in MongoDB:**

   In the cookbook: "ETL-Process with datasets", we learned how to model, import and index datasets.

The Purpose of this document, is to provide information related to creating a secure web server with ModSecurity. The web server we will be using is a NGINX web server.

We start of by discussing the basics of Docker, ModSecurity and the OWASP Core rule set.

Afterwards we will create a web application with Python-Flask and Gunicorn. This web application will eventualy be hosted on the secure Nginx web server.

After the creation of the web application we will start with the creation of the Nginx web server and the implementation of the OWASP core rule set. Al the steps will be explained in detail with images to give a visual representation of what happens.

The web application and the web server will both be running in separate Docker containers, which can easily be managed in the docker-compose file.

The code that will be created during this cookbook is in the folder "scripts" in the following files:

Root folder files	Flask-Gunicorn folder files	Nginx-Modsecurity folder files
docker-compose.yml	app.py	index.html
create_file_structure.sh	index.html	Dockerfile
	style.css	nginx.conf (chapter 8)
	Dockerfile	nginx.conf (chapter 9)
	requirements.txt	

A special thanks to the following organizations and people there:

- **Trustwave SpiderLabs for the immense work they put into the ModSecurity project.**
- **Docker for providing the software.**
- **Nginx for providing the lightweight web server.**

Table of Contents

Tabel of Illustrations

1 Introduction to Docker

Docker is a helpful tool for packing, shipping and running applications within "containers". In this chapter I will explain the fundamental concepts around what a "container" is and how it compares to a Virtual Machine (VM).

Illustration 1: Docker logo

1.1 Containers and Virtual Machines

Containers and VM's are similar in their goals: to isolate an application and its dependencies into a self-contained unit that can run anywhere. Moreover, containers and VM's remove the need for physical hardware, allowing for more efficient use of computing resources, both in terms of energy consumption and cost effectiveness. The main difference between containers and VMs is in their architectural approach. (Kasireddy, Free Code Champ, 2019)

1.1.1 Virtual Machines

A VM is essentially an emulation of a real computer that executes programs like a real computer. VM's run on top of a physical machine using a "hypervisor".

A hypervisor is a piece of software, firmware, or hardware that VMs run on top of. The hypervisors themselves run on physical computers, referred to as the "host machine". The host machine provides the VMs with resources, including RAM and CPU. These resources are divided between VMs and can be distributed as you see fit. So if one VM is running a more resource heavy application, you might allocate more resources to that one than the other VMs running on the same host machine. (Kasireddy, Free Code Champ, 2019)

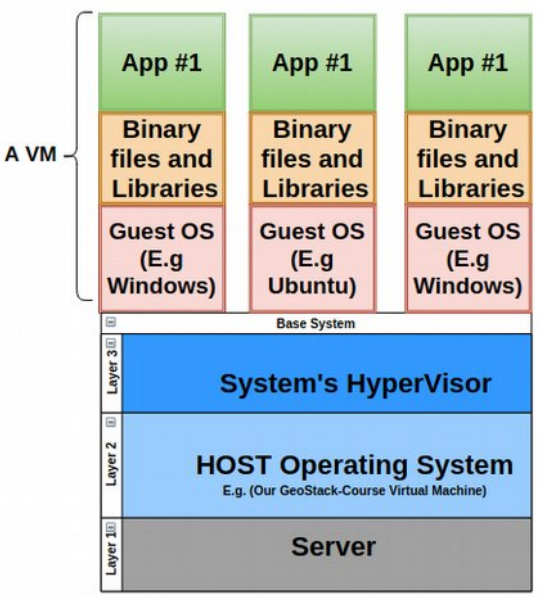


Illustration 2: VM Layout diagram

1.1.2 Docker containers

Unlike a VM which provides hardware virtualization, a container provides operating-system-level virtualization by abstracting tools and packages needed for the desired software to run. The one big difference between containers and VM's is that containers share the host system's kernel with other containers.

A few advantages of using docker are as follows:

- **Ease of use:** Docker allows anyone to package an application on their laptop, which in turn can run unmodified on any system.
- **Speed:** Containers are very lightweight and fast. You can create and run a docker container in seconds.
- **Docker Hub:** Docker users also benefit from the increasingly rich ecosystem of Docker Hub, which you can think of as an "app store for Docker images."
- **Modularity and Scalability:** Docker makes it easy to break out your application's functionality into individual containers.

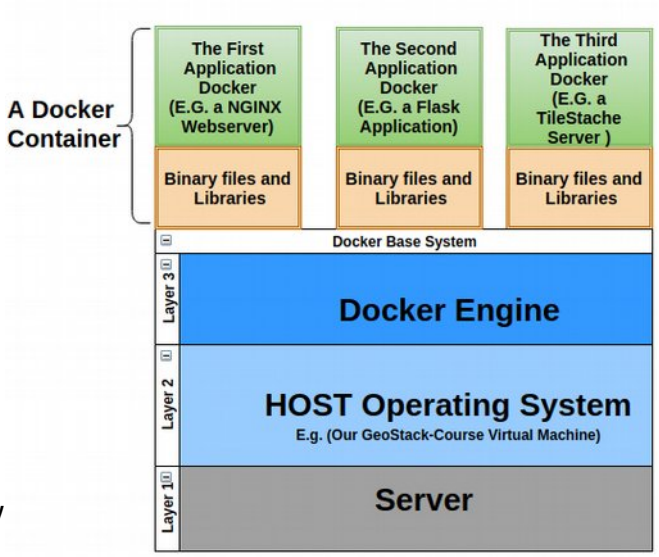


Illustration 3: Docker Container layout

1.2 How does Docker work?

Docker exists out of multiple parts. This section will describe each part globally. In illustration 4 you can find a visual representation of the Docker infrastructure.

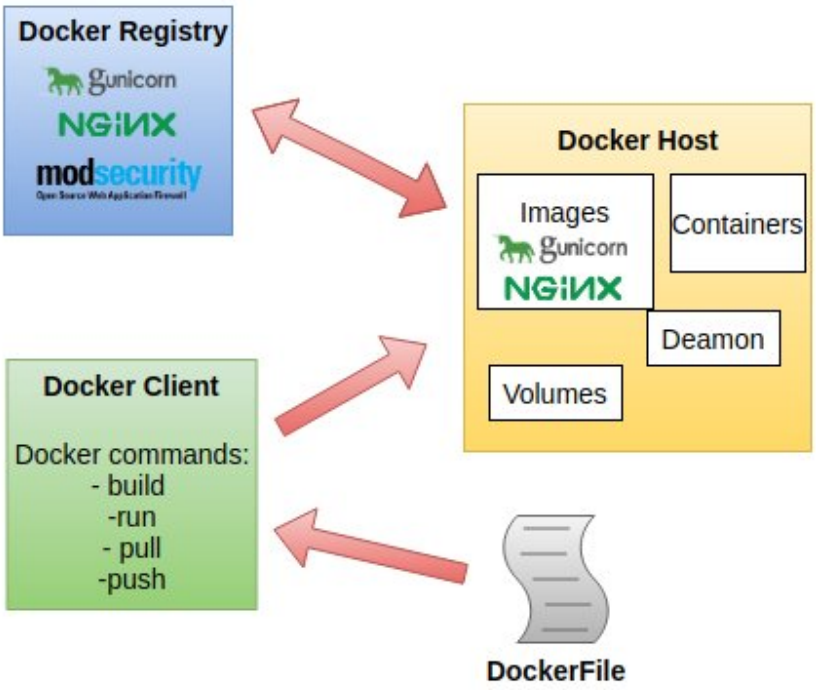


Illustration 4: Docker Infrastructure visualization

1.2.1 Docker Engine

The Docker engine is the layer on which Docker runs. It's a lightweight run-time tool that manages containers, images, builds, and more. It runs natively on Linux systems and is made up of the following components:

- A Docker Daemon that runs in the host computer.
- A Docker Client that then communicates with the Docker Daemon to execute commands.
- A REST API for 3interacting with the Docker Daemon remotely.
- The images created by the user. Each image has a base image which can be obtained from the Docker registery. The image can be used to run as Docker container.

1.2.2 Dockerfile

A Dockerfile is the file in which you write the instructions required to build your own Docker image. Examples of such instructions are as follows:

- "RUN apt-get y install {package-name}" To install certain packages in a docker container.
- "EXPOSE 8000" To set the port on which the Docker container will be available.
- "ENV ANT_HOME /usr/local/apache-ant" to pass an environment variable to the Docker container.

1.2.3 Docker Images and Containers

Images are read-only templates that you build from a set of instructions written in your Dockerfile. Images define both what you want in your packaged application and its dependencies to look like and what processes to run when it's launched.

A Docker container wraps an application's software into an invisible box with everything the application needs to run. That includes the operating system, application code, runtime, system tools, system libraries, and etc. Docker containers are built off Docker images. Since images are read-only, Docker adds a read-write file system over the read-only file system of the image to create a container. (Kasireddy, Free Code Champ, 2019)

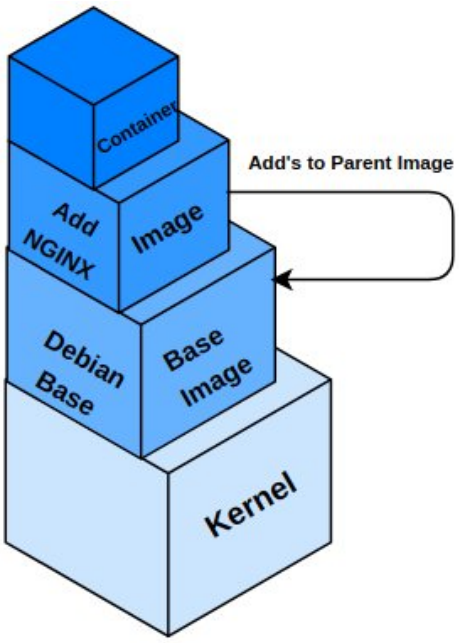


Illustration 5: Docker image layout

1.2.4 Docker-Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose you use a YAML file to configure your application's services. Then with a single command you create and start all the services from your configuration file. The illustration below shows a part of the Docker compose file (called: "docker-compose.yml") which we are going to create during the cookbook. When this docker-compose file is run the Flask docker container (called:"flask_app") will start. But more on this later!

```
version: '3.7'

services:

  # Set the name of the service
  flask_app:

    # Set the container name
    container_name: flask_app

    # Restart when container goes down
    restart: always

    # Set build location. This is
    # where the dockerfile is located
    build: ./flask-gunicorn

    # Set the ports of the container
    # If one is set it's the container port
    # If two are set it's:
    # {HOST}:{CONTAINER}
    ports:
    - "8000"

    # Set the command wich runs on startup
    command: gunicorn -b :8000 app:app

    # Set volume of the container.
    # This volume is the downloads folder in our
    # flask-gunicrion folder.
    volumes:
    - .:/downloads
```

Illustration 6: docker-compose example

2 Introduction to Flask with Gunicorn

Illustration 7: Logo Flask

Flask is a Python WSGI web application framework which is very lightweight. It's also known as an micro framework since it's so small and doesn't need a lot of configuration to get it up and running. It is especially design to get started quick and easy, with the ability to scale up to a complex application.

The Flask micro framework has no database abstraction layer, form validation, or any other components found in third-party libraries. As you can see in illustration 8 , it requires a minimal amount of code to write a simple web application.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

Illustration 8: Flask application code

Flask supports extensions that can add application featur3es as if they were implemented in Flask itself. A few examples of such extensions are as follows:

- **Object-relational mapping**
- **form validation**
- **upload handling**
- **Open authentication technologies**

As mentioned above, Flask is a WSGI web application framework. WSGI stands for Web Server Gateway Interface. WSGI is by design a simple standard interface for running Python code.

Illustration 9: Logo Gunicorn

For the communication between our Flask web application and our Nginx web server to run smoothly, we need a WSGI server. For this we are going to use Gunicorn, also known as "Green Unicorn". Gunicorn is used to translate incoming data into a readable format for NGINX.

In illustration 10 you can see a visual representation of how a WSGI server works. Again, the WSGI server we are going to be using is Gunicorn and as web server we are going to use Nginx.

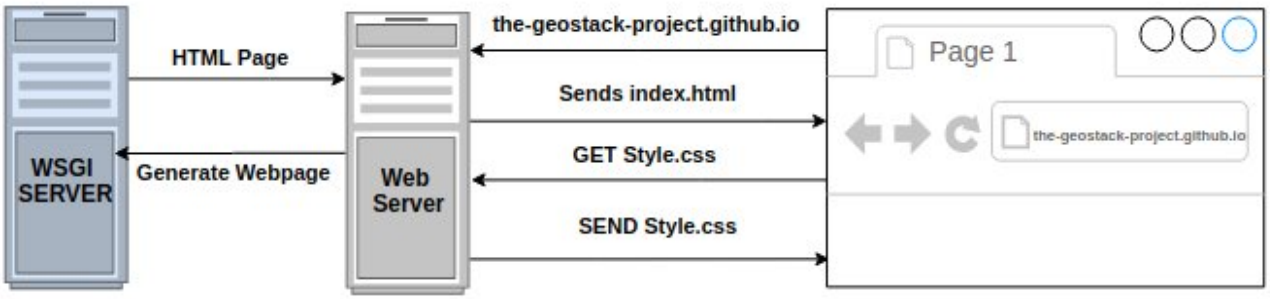


Illustration 10: Dataflow Diagram of webserver with WSGI

Our web server is configured to pass requests to the WSGI container, which runs our web application, and then pass the response (in the form of HTML) back to the requester.

3 Introduction to NGINX

Illustration 11: Logo NGINX

Nginx, pronounced "engine X", is a fast and lightweight web server, that can be used to serve static files, but is often used as a reverse proxy. It has some very nice features like load balancing and rate limiting. In this chapter I will explain why it's useful to use Nginx. I will also show some simple examples on how to serve static files and create a reverse proxy. We will need this later on in the document.

3.1 Why use NGINX

Nginx is commonly used if you need a lightweight web server with:

- **A reverse proxy**
  A reverse proxy is useful if you have multiple web services listening on various ports and you need to reroute requests internally. This allows you to run multiple domain names on port 80 which is the port the web server is running on.
- **Load balancing**
  Load balancing is a technique used to distribute workload across multiple machines. Load balancing could be division of processes, hard drives and other resources.
- **Rate limiting**
  Rate limiting is kind of the same as load balancing, but in the case of rate limiting the work load is divided among users of the web server.

3.2 Serve static files

The most basic task for a web server is serving static files. Static files could be html files, css files, txt files etc. To serve these types of files you just copy a .txt, .html, .zip, or any other kind of file in the root directory of the web server and the server will return them directly.

The configuration example in illustration 12 will listen on port 80 and can be viewed at http://localhost:80/.

Also included in illustration 12, is an example of setting separate controls on a sub directory by turning off directory listing and designating specific index files only. It will also serve the **/var/www/static-content/** directory on the URL of **/static/**.

```
http {
    server {
        # Here we define the port on which the server listens.
        listen       80;
        # Here we define the root folder of the NGINX webserver.
        root /path/to/public/dir;
        # Here we define the default location. The index.html
        # is passed when it's accessed.
        location / {
            index  index.html index.htm;
        }
        # Here we define the location on which the static
        # content is accesible.
        location /static-content/ {
            alias /var/www/static-content/;
        }
    }
}
```

Illustration 12: Serve static files from NGINX

3.3 Create reverse proxy

The configuration example seen in illustration 13 sends traffic from port 80 to localhost:5000, including the original IP as an extra HTTP header. Some proxies use X-Forwarded-For instead.

```
http {
    server {
        # Here we define the port on which the server listens.
        listen       80;
        # Here we define the domain name of the server.
        server_name the-geostack-project.com
        # Here we define the default location. When it's accessed localhost:5000 is proxyd.
        location / {
            proxy_pass http://localhost:5000/;
        }
    }
}
```

Illustration 13: Creating a proxy in NGINX

4 Introduction to ModSecurity

ModSecurity is a toolkit for real-time web application monitoring, logging, and access control. I like to think about it as an enabler: there are no hard rules telling you what to do; instead, it is up to you to choose your own path through the available features. (Trustwave SpiderLabs, 2019)

Illustration 14: Logo ModSecurity

Some ModSecurity features are as follows:

- **Full HTTP traffic logging:** These day's web servers do not do a lot when it comes to logging for security purposes. ModSecurity enables you to log anything you need, including raw transaction data for forensics.
- **Continuous passive security assessment** Continuous passive security assessment is a variation of real-time monitoring, where, instead of focusing on the behavior of the external parties, you focus on the behavior of the system itself. It's an early warning system of sorts that can detect traces of many abnormalities and security weaknesses before they are exploited. (Trustwave SpiderLabs, 2019)
- **Web application hardening** One of the best features of ModSecurity is attack surface reduction, in which you selectively narrow down the HTTP features you are willing to accept (e.g., request methods, request headers, content types, etc.)
- **Something small, yet very important to you** Real life often throws unusual demands to us, and that is when the flexibility of ModSecurity comes in handy where you need it the most. It may be a security need, but it may also be something completely different. For example, some people use ModSecurity as an XML web service router, combining its ability to parse XML and apply XPath expressions with its ability to proxy requests. Who knew? (Trustwave SpiderLabs, 2019)

The ModSecurity project lives by 4 guiding principles. These principles are as follows:

- **Flexibility:** ModSecurity was originally created for the creator himself. He wanted to be able to intercept, analyze and store HTTP traffic. He didn't see any value in hard coded functionality.

  Because of this he wanted to achieve flexibility by giving us a powerful rule language, which allows you to do exactly what you need to, in combination with the ability to apply rules only where we need to. (Trustwave SpiderLabs, 2019)
- **Passiveness:** ModSecurity Takes great care to never interact with a transaction unless you configure it to do so. The reason for this is because the owner does not trust third-party tools.
- **Predictability:** The creator of ModSecurity states that there is no such thing as a perfect tool. On the website you can find a list of all weaknesses and how to work around them.
- **Quality over quantity:** Since the beginning of ModSecurity 6 years ago the developers came up with a lot of ideas for extra functionality. Most of them aren't implemented yet because they have limited resources at their disposal. They chose to limit the functionality of ModSecurity, but they do really well at what they decided to keep in.

5 Introduction to the OWASP CRS

The **OWASP ModSecurity Core Rule Set (CRS)** is a set of generic attack detection rules for use with ModSecurity or compatible web application firewalls. The CRS aims to protect web applications from a wide range of attacks, including the OWASP Top Ten, with a minimum of false alerts. (Trustwave SpiderLabs, 2019)

Illustration 15: Logo OWASP CRS

The Core Rule Set provides protection against many common attack categories, including:

- ✔ **SQL Injection** : This is a code injection technique that is able to destroy your database. It's one of the most common web hacking techniques. The goal of this attack is to place malicious code in SQL statements, via web page input.
- ✔ **Cross Site Scripting (XSS)**: This is a type of vulnerability where the attacker injects client side scripts into the web page viewed by other users. This could enable the attacker to bypass access controls.
- ✔ **Local / Remote file Inclusion** : The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a "dynamic file inclusion" mechanisms implemented in the target application.
- ✔ **Remote Code Execution**: An attacker has access to you device and can make changes and execute code from a distance.
- ✔ **PHP Code Injection:** allows the attacker to insert malicious PHP code straight into a program/script from some outside source.
- ✔ **HTTP Protocol Violations:** These types of attacks include Denail of Service attacks, Addressing attacks, Authentication attacks and Session management attacks.
- ✔ **HTTProxy:** This type of attack can give the attacker the possibility to proxy the outgoing HTTP requests.
- ✔ **Shellshock:** This attack could enable an attacker to cause Bash to execute arbitrary commands and gain unauthorized access to many Internet-facing services, such as web servers.
- ✔ S**ession Fixation:** These attacks attempt to exploit the vulnerability of a system that allows one person to fixate (find or set) another person's session identifier.
- ✔ **Metadata / Error Leakages:** The OWASP CRS makes sure no metadata or error leakages occur. This makes it a lot harder for an attacker to find any bugs in our web server.
- ✔ **Project Honeypot Blacklist**: A project honeypot is software that is embedded in a web site that collects information about IP addresses and email addresses. If these turn out to be malicious, the software puts them on a so called black list.
- ✔ **GeoIP Country Blocking:** This is a technology that restricts access to Internet content based upon the user's geographical location. When using a VPN this feature can be bypassed.

6 The folder and file structure

In this chapter I will explain what the purpose is of each file and folder in the file structure. The root folder structure is show in illustration 16.

6.1 The root folder

As you can see the name of the root folder is 20191015_Secure_Webserver_Nginx. In this folder, we have 1 file and 2 sub-folders:

```
20191015_Secure_Webserver_Nginx
    ├── docker-compose.yml
    ├── flask-gunicorn
    └── nginx-modsecurity
```

Illustration 16: root folder structure

1) **File: docker-compose.yml**

   In this file we will define services, networks and volumes. In our case we have 2 services:
   1) flask-gunicorn
   2) nginx-modsecurity

2) **Folder: flask-gunicorn**

   This folder is for the code and files related to the flask-gunicorn container.

3) **Folder: nginx-modsecurity**

   This folder is for the code and files related to the nginx-modsecurity container.

6.2 The flask-gunicorn folder

The flask-gunicorn folder structure is shown in illustration 17.

As you can see the name of the folder is flask-gunicorn. In this folder, we have 3 files and 3 sub-folders:

```
20191015_Secure_Webserver_NGINX
├── docker-compose.yml
├── flask-gunicorn
│   ├── app.py
│   ├── dockerfile
│   ├── downloads
│   ├── requirements.txt
│   ├── static
│   │   ├── css
│   │   │   └── style.css
│   │   └── img
│   │       └── favicon.ico
│   └── templates
│       └── index.html
└── nginx-modsecurity
    ├── Dockerfile
    ├── nginx.conf
    └── static
        └── index.html
```

Illustration 17: flask-gunicorn structure

1) **File: App.py**

   In this file we define the logic of our flask web application.

2) **File: Dockerfile**

   In this file we define the commands to assemble the docker image.

3) **File: requirements.txt**

   In this file we define the modules that are required to run the Flask application.

4) **Folder: downloads**

   In this folder we place all the downloadable files.

5) **Folder static**

   In this folder we place all the static files. This folder has 2 sub-folders:
   - **css**
     In this folder we place the style sheets for our web application.
   - **img**
     In this folder we place the images of our web application.

6) **Folder: templates**

   In this folder all the the HTML files of our app are located.

6.3 The nginx-modsecurity folder

The nginx-modsecurity folder structure is shown in illustration 18.

As you can see in the illustration below the name of the folder is nginx-modsecurity. In this folder, we have 3 files and 3 sub-folders:

1) **File: Dockerfile**

   In this file we define the commands to assemble the docker image.

2) **File: nginx.conf**

   In this file we will create our custom Nginx configuration.

3) **Folder: static**

   In this folder we will put all the static files related to the Nginx web server.

In illustration 18 you can see what the final file structure will look like once we finished creating the file structure .

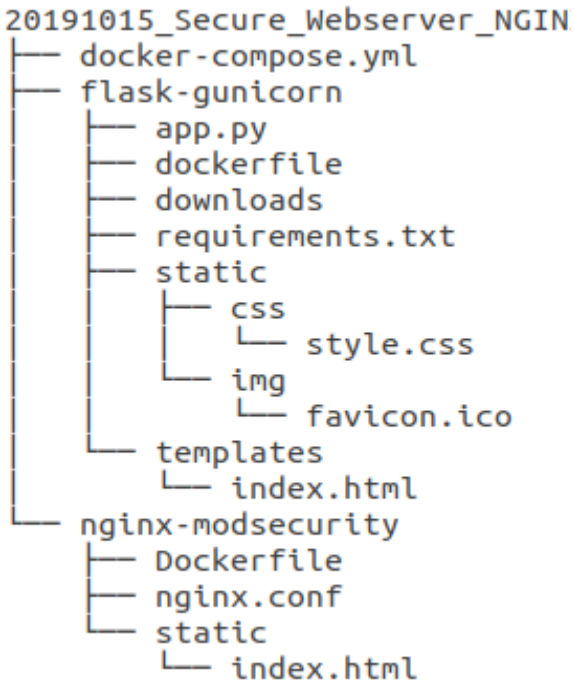


Illustration 18: Total file structure

Now that you know the purpose of each file and folder in the file structure, we can start with the creation of the files and folders. In the next chapter I will discuss how this is done.

6.4 Creating the file structure

Now let's create the basic file structure We are going to do this by creating a bash script which creates the files and folder required in this cookbook. This chapter describes how to setup the bash script and what happens in this bash script. By creating such a bash script you can make it easy for other users to create the exact desired file structure of a project.

Some information about the script can be found in the illustration below.

```
#!/bin/bash
# Bash script for creating folder structure of Secure NGINX Web server
# This script is used in the Cookbook recipe
# "A secure NGINX webserver with modsecurity"
#-------------------------------------------------------------------------------
```

Illustration 19: Bash script information

First we need to create the script. This is done by running the following command in a terminal:

```
touch create_file_structure.sh
```

Now let's open the newly create file and start coding. We will start of by creating the code which creates the project directory. This is done by assigning the name of our directory to variable "dir" as shown in the illustration below.

```
#-------------------------------------------------------------------------------
#0) Set variables

#dir variable represents the main directory
dir=`date +%Y%m%d_Secure_Webserver_NGINX`

#Create a new main directory with the current date
mkdir $dir

#-------------------------------------------------------------------------------
#1) Create the docker-compose.yml
#This file is used
touch $dir/docker-compose.yml
```

Illustration 20: Set variables

Now we have to create the flask-gunicorn folder and files. How this is done is shown in the illustration below.
The command mkdir is used to create a new directory. The command “touch” is used to create a new file.

```
#-------------------------------------------------------------------------------
#2) Create the flask-gunicorn directories.
mkdir $dir/flask-gunicorn
mkdir $dir/flask-gunicorn/templates
mkdir $dir/flask-gunicorn/downloads
mkdir $dir/flask-gunicorn/static
mkdir $dir/flask-gunicorn/static/css
mkdir $dir/flask-gunicorn/static/img

#-------------------------------------------------------------------------------
#3) Create the flask-gunicorn files.
touch $dir/flask-gunicorn/app.py
touch $dir/flask-gunicorn/dockerfile
touch $dir/flask-gunicorn/requirements.txt
touch $dir/flask-gunicorn/templates/index.html
touch $dir/flask-gunicorn/static/css/style.css
touch $dir/flask-gunicorn/static/img/favicon.ico
```

Illustration 21: Create flask-gunicorn directories and folders

Now we want to do the same for the the nginx-modsecurity folder. First we use the “mkdir” command to create the required folders and then we use the “touch” command to create the required files. How this is done is shown in the illustration below.

```
#-------------------------------------------------------------------------------
#4) Create the nginx-modsecurity directories.
mkdir $dir/nginx-modsecurity
mkdir $dir/nginx-modsecurity/static

#-------------------------------------------------------------------------------
#5) Create the nginx-modsecurity files.
touch $dir/nginx-modsecurity/Dockerfile
touch $dir/nginx-modsecurity/nginx.conf
touch $dir/nginx-modsecurity/static/index.html
```

Illustration 22: Create nginx-modsecurity files and folders

That’s it! You created the bash script required to create the file structure required in this cookbook.

To run this bash script, enter the command “sudo bash ./create_file_structure.sh ” in a terminal. The path of the terminal should be set to the location of the bash script.

7 Setting up Flask and Gunicorn

Let's start off with creating the Flask web application.

7.1 Creating the requirements.txt file

As discussed in the previous chapter we have a text file called requirements.txt to include all the required Python modules for the project.

The 2 required modules for this project are as follows:

- flask
- gunicorn

So let's open the file and add flask and gunicorn, as shown in illustration 23.

Illustration 23: Add required modules

7.2 Creating the app.py file

Now that we have defined the requirements of our web application, we need to create the main logic of our web application. This is done in the app.py file.

We start off with importing the required modules. This is shown in illustration 24.

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 0) Import the required modules or functions from modules.
from flask import Flask, render_template
# Flask is used to create a WSGI application object called 'app'
# render_template is used to serve static HTML files from the templates folder.
```

Illustration 24: Import required modules

Now that we have imported the required modules we need to create an instance of the web server application. How this is done is shown in illustration 25.

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1) Create a WSGI webserver application with the script file name (__name__).
#    Note: passing __name__ as a parameter makes it possible for the app object
#          to determine if this script was started as an application or if it
#          was imported as a module.
#          For the same reason the application part of the script is started
#          with a "if __name__ == '__main__':" statement but now for Flask with
#          the app.run() function instead of the usual main() function.
app = Flask(__name__) # app is now an instance of the Flask class (= WSGI app!).
```

Illustration 25: Create Flask instance

Now that we have created an instance of the Flask class, we need to create the main function called: index. This function is triggered when navigating to the URL on which our application is running (e.g. http://localhost:5000/).
We'll create the index.html later in this document so do not worry about that yet. How to create the main function is shown in illustration 26.

```
@app.route('/') # Define the URL to bind to the function!
def index():
    # This is the associated function that runs when an URL is received
    # Define message texts and pass these as a parameter to render_template().
    # render_template() looks in the app/templates folder for index.html!
    # The HTML file uses a CSS file style.css that is in the /static/css folder.
    message_title =    '--> Welcome to a secure web application! <--'
    message_subtitle = '---Running on a NGINX Web server---'
    return render_template('index.html',\
                           message_title=message_title,\
                           message_subtitle=message_subtitle)
```

Illustration 26: Create index function

As you can see we defined 2 variables:

- message_title
- message_subtitle

These variables will be passed to the index.html file when the function is triggered. These variables will be displayed as messages on the web page.

Now we have to make sure the function app.run() is triggered when we navigate to the URL on which the web application is running. How this is done, is shown in illustration 27.

```
if __name__ == '__main__':
    # Call the trigger function app.run() to run the Flask application webserver.
    app.run(host='0.0.0.0') # Run the Flask WSGI app in the normal operation mode!
    #app.run(debug=True) # Run the Flask WSGI app in debugging mode!
```

Illustration 27: Call the trigger function app.run()

7.3 Creating the main template file: index.html

Now that we have the main logic of our Flask web application we need to create the page that will be rendered when the index function is triggered. So let's open the index.html file located in the templates folder.

First we want to define the document type of our document, which is HTML in this case. This is shown in illustration 28.

```
<!-- templates/index.html for the Secure NGINX web server -->
<!-- Define the type of our document-->
<!DOCTYPE html>
<html>
```

Illustration 28: Define document type

Now we want to define the content of the head element. In this element we define the scripts, styles and meta information etc. How this is done, is shown in illustration 29.

```
<!-- The <head> element can include a title for the document,
scripts, styles, meta information, and more.-->
<head>

    <!-- Set the title of our web page. -->
    <title>--> Secure Nginx Webserver</title>

    <!-- Define the stylesheet that will be used.
    The href variable is the location of the style sheet
    /static/css/style.css in this case.-->
    <link rel="stylesheet" href="/static/css/style.css">
</head>
```

Illustration 29: Define head tag

Now that we have defined the content of the head tag, we want to do the same for the body tag. The body contains all elements of a HTML document, such as text, hyperlinks and images. In the body we also call the variables we set in the index function of the app.py file. How this is done is shown in illustration 30.

```
<body>
    <div>

        <!-- The <header> element represents a container
        for introductory content or a set of navigational links.
        In this case in contains the message_title passed
        in the index()function located in our app.py-->
        <header class="title_bold">
            {{message_title}}
        </header>

        <!--The <p> tag defines a paragraph.
        In this case it contains the message_subtitle passed
        in the index() function located in our app.py-->

        <p class="subtitle_bold">
            {{message_subtitle}}
        </p>
    </div>
</body>
```

Illustration 30: Define body tag

7.4 Creating the style sheet file: style.css

Now that we created our index.html we need to create our style sheet. This is done in the style.css file located in the folder /static/css/.

We want to start of with defining the style of the body element. This is shown in illustration 31.

```
/* static/css/style.css */

/*
Define the style of the body
*/
body {
    background: grey;
}
```

Illustration 31: Style body element

Next we want to define the style of some basic elements such as the header element and the list element. How this is done is shown in illustration 32.

```
/*
The <h1> to <h6> tags are used to define
HTML headings.
<h1> defines the most important heading.
<h6> defines the least important heading.

We use color to define the text color.
We use font-size to define the size of
the text
*/

h1 {
  color: black;
  font-size: 2em;
}

h2 {
  color: black;
  font-size: 1.25em;
}

h3 {
  color: black;
  font-size: 1em;
}

/*
The <li> tag defines a list item.
The <li> tag is used in ordered lists(<ol>),
unordered lists (<ul>), and in menu lists (<menu>).
*/

li {
  color: blue;
}
```

Illustration 32: Style basic elements

7.5 Setting up the Dockerfile

The Dockerfile will contain the commands we need to run to assemble the docker image for the Flask web application. The Dockerfile is located in the flask-gunicorn folder.
We want to start by defining the base image on which we are going to build our own image. How this is done, is shown in illustration 33.

```
# Get the base image with the OS and Python from DockerHub
FROM python:3.7.2
```

Illustration 33: Define base image

Next want to create a directory in our docker container in which the Flask web application code will be located. We also want to set this directory as work directory. This makes sure all the commands that follow, after this line of code, will be executed in the work directory.

```
# Create the image webserver folder including the parent folders
# project and home because they do not exist yet by using -p.
RUN mkdir -p /home/project/flask_app

# Set the new folder as the default working directory
WORKDIR /home/project/flaks_app
```

Illustration 34: Set work directory

Now that we set the work directory, we want to copy the requirements file, we created earlier, to that directory. After we copied the requirements.txt file we want to execute it and thus install the required modules in our docker container. How this is done, is shown in illustration 35.

```
# Copy the Python requirements.txt file into the webserver folder.
# This config file defines the gunicorn and flask modules to be installed.
COPY requirements.txt /home/project/flask_app
# Run the requirements.txt file with pip to install gunicorn and flask.
RUN pip install --no-cache-dir -r requirements.txt
```

Illustration 35: Install required modules

Finally we want to copy the contents of the flask-gunicorn folder to our project folder in the docker container. How this is done, is shown in illustration 36.

```
# Finally copy the gunicorn-flask folder into the image webserver folder.
# This copies the subfolders and the flask webserver WSGI app file app.py
COPY . /home/project/flask_app
```

Illustration 36: Copy folder to project directory

Now that we have finished the docker file, we want define the command that will run the flask web application on the gunicorn web server. This is shown in illustration 37.

```
# Run de flask app app.py on the gunicorn WSGI webserver on port 8000.
CMD ["gunicorn"  , "-b", "0.0.0.0:8000", "app:app"]
#RUN gunicorn -b 0.0.0.0:8000 app:app
```

Illustration 37: Set run command

That's it! We have finished constructing our docker file. In the next chapter is explained how to run the Flask web application in a docker container.

7.6 Running the web application as docker container

To run the Flask web application in a docker container, we first need to build the container. This is done by opening a command prompt and entering the following command: “docker build -t flask-gunicorn -f Dockerfile .” as shown in illustration 38.

The -t argument in this command, is the name the container gets.
The -f argument in this command, is the docker file which we are going to use to build the image.

```
docker build -t flask-gunicorn -f Dockerfile .
```

Illustration 38: Build flask-gunicorn

The output of the command should be similar to the output shown in illustration 39.

```
Successfully built f6cc72c2a7d2
```

Illustration 39: Output build command

Now that we have successfully build the flask-gunicorn image, we want to run it. This is done by entering the command: “docker run -ti --rm -p 8000:8000 flask-gunicorn”, as shown below.

```
docker run -ti --rm -p 8000:8000  flask-gunicorn
```

Illustration 40: Run flask-gunicorn

Some explanation for this command is as follows:

- **-ti** indicates that we want do not want to run the docker container in the background, now we can see the log’s of the docker container.
- -**p** indicates the ports on which the container will be running. The 80 on the left of the :, means that the container is available on port 80 on the host. The 80 on the right of the :, means that the container is available on port 80 in the docker container.
- **-rm** makes sure the docker container is removed once it stops.

The output of this command will look similar to the output shown in illustration 41.

```
[2019-10-17 20:13:06 +0000] [1] [INFO] Starting gunicorn 19.9.0
[2019-10-17 20:13:06 +0000] [1] [INFO] Listening at: http://0.0.0.0:8000 (1)
[2019-10-17 20:13:06 +0000] [1] [INFO] Using worker: sync
[2019-10-17 20:13:06 +0000] [8] [INFO] Booting worker with pid: 8
```

Illustration 41: Output run command

Some explanation concerning the output is as follows:

- The first line of the output indicates that the Gunicorn web server is starting.
- The second line of the output shows the Gunicorn web server runs on http://0.0.0.0:8000.
- The third line of the output indicates that the Gunicorn worker type is set to sync. This means that Gunicorn delegates one process for each request.
- The last line of the output indicates that the Gunicorn worker started with process id 8.

When we navigate to http://0.0.0.0:8000 we are presented with the index.html we created earlier in this document. The landing page is shown in illustration 42.

Illustration 42: Landing page

8 Setting up NGINX, ModSecurity and the CRS

Now that we finished setting up our Flask web application and our Gunicorn web server, we are going to start constructing the NGINX container of our secure web server.

A good thing about the way we are going to install the Core rule set, is that every time we rebuild the docker image the new rules are added to our rule set. This means our web application defense is always up to date.

8.1 Creating the index.html

Just like in chapter 7, we want to create a custom HTML page. This page is used to test whether our Nginx web server is correctly configured. For this reason, we are going to keep this index.html simple without any styling.

We want to start with defining our document type and our head tag . Since we are not going to use any styling for this index.html, we do not need a reference to a style sheet in our head. We only need to define a title. How this is done, is shown in illustration 43.

```
<!-- static/index.html for the Secure NGINX web server -->
<!-- Define the type of our document-->
<!DOCTYPE html>
<html>
    <!-- The <head> element can include a title for the document,
    scripts, styles, meta information, and more.-->
    <head>
        <!-- Set the title of our web page. -->
        <title>Secure Nginx Webserver</title>
    </head>
```

Illustration 43: Define head

Now that we have set the head tag and the document type we can create the content of our body. How this is done, is in shown in illustration 44.

Illustration 44: Define body

Now we need to close all our tags. How this is done, is shown in illustration 45.

```
            </p>
        </div>
    </body>
</html>
```

Illustration 45: Close tags

That's it, we now have created our custom index.html for our Nginx web server.

In the next chapter we'll create a Nginx configuration file, to make sure our custom index.html file is presented to use when loading the web server.

8.2 Creating the Nginx configuration file

Because we are using ModSecurity, we have to create a custom Nginx configuration. As mentioned in chapter 6, we can do this in the nginx.conf file.

We start of with loading the nginx-modsecurity connector module. We do this at the top of our nginx.conf file. How this is done, is shown in illustration 46.

```
# Load the modsecurity connector module.
load_module modules/ngx_http_modsecurity_module.so;
```

Illustration 46: Load connector module

Now that we have loaded the required module, we need to set the user to root, we do this because we need to elevate the Nignx privileges to root . How this is done, is shown in illustration 47.

```
# Set the user to root
user  root;
```

Illustration 47: Elevate privileges

Now we have to set the configuration related to the speed of the NGINX web server. How this is done, is shown in illustration 48.

```
# Set the amount of processes
# A worker process is a single-threaded process.
# If Nginx is doing CPU-intensive work such as SSL or
# gzipping and you have 2 or more CPUs/cores
# then you may set worker_processes to be equal to the
# number of CPUs or cores.
worker_processes  1;


events {
    # The worker_connections and worker_processes from the main section
    # allows you to calculate max clients you can handle:
    # max clients = worker_processes * worker_connection
    worker_connections  1024;
}
```

Illustration 48: Configure processes

Now we have to enable ModSecurity and point the configuration file to the file that is going to contain our ModSecurity rules. We also need to point our root folder to our custom index.html file. How this is done, is shown in illustration 49.

```
http {

    # Toggle Modsecurity on
    modsecurity on;

    # Set the location of the modsecurity rules configuration file.
    # This is the file we copied the rules to in the Dockerfile :
    # " /etc/modsecurity.d/owasp-crs/rules/*.conf" >> /etc/modsecurity.d/include.conf "
    modsecurity_rules_file /etc/modsecurity.d/include.conf;

    server {
        #When navigating to localhost/ it returns our costum made index.html
        location / {
            # We have to set the root folder to "/data/Publishers/" since we
            # copied our static folder,that includes our costum index.html,
            # to the "/data/Publishers/" in our docker container.
            root   /data/Publishers/;

            index  index.html index.htm;
        }
    }
}
```

Illustration 49: Toggle ModSecurity and configure server

8.3 Setting up the Dockerfile

First we want to pull the base image on which we are going to build our own image. How this is done, is shown in illustration 50.

```
# 1) Get this base image with the OS and the nginx webserver from DockerHub.
# Original OWASP image with SETPROXY=True
```

Illustration 50: Define base image

Now that we have pulled the base image, we set the arguments which will be used throughout the Dockerfile. How this is done, is shown in illustration 51. First we set the required arguments to select the github repository and to download (git-pull) the desired Core Rule Set (CRS) branch and commit version. You can change the arguments, to the desired branch or commit version.

```
# 2) Set arguments

Set repo branch argument.
The branch that will be fetched and used
RG BRANCH=v3.3/dev

Set branch commit argument.
The commit version that will be downloaded and used
RG COMMIT=v3.3/dev

Set repository argument.
This is the official OWASP CRS Github repository.
RG REPO=SpiderLabs/owasp-modsecurity-crs
```

Illustration 51: Set arguments

Now that we set the arguments, we want to make sure the Core Rule Set (CRS) is installed in our Nginx web server. The core rule set can be cloned from the Github repository located at: https://github.com/SpiderLabs/owasp-modsecurity-crs. We are going to clone the repository and then copy the rule set to the configuration file which contains all our rules. How this is done, is shown in illustration 52.

```
# Update the package index
RUN apt-get update && \

  # Install required libraries
  apt-get -y install python git ca-certificates iproute2 && \

  # Create the directory in which the repo will be cloned
  mkdir /opt/owasp-modsecurity-crs-3.2 && \

  # Navigate to that directory
  cd /opt/owasp-modsecurity-crs-3.2 && \

  # Initialize a github repository
  git init && \

  # Add the Modsecurity Core Rule Set repository.
  # The repository was set in step 2.
  git remote add origin https://github.com/${REPO} && \

  # Fetch the branch set in step 2
  git fetch --depth 1 origin ${BRANCH} && \

  # Dowload the commit version.
  # This argument was set in step 2.
  git checkout ${COMMIT} &&\

  # Copy the contents of the example configuration
  # file to the actual config file.
  mv crs-setup.conf.example crs-setup.conf && \

  # Create a link between owasp-modsecurity-crs-3.2  and owasp-crs file.
  ln -sv /opt/owasp-modsecurity-crs-3.2 \
  /etc/modsecurity.d/owasp-crs && \

  # Copy the rulset from the cloned github repository to the
  # modsecurity configuration file
  printf "include /etc/modsecurity.d/owasp-crs/crs-setup.conf\ninclude \
  /etc/modsecurity.d/owasp-crs/rules/*.conf" >> \
  /etc/modsecurity.d/include.conf && \

  # Toggle on Modesecurity rule Engine in the modesecurity configuration file.
  # Change the SecRuleEngine directive in the configuration to change from the
  # default “detection only” mode to actively dropping malicious traffic.
  sed -i -e 's/SecRuleEngine DetectionOnly/SecRuleEngine On/g' \
  /etc/modsecurity.d/modsecurity.conf
```

Illustration 52: Install OWASP CRS

Now that we made sure the CRS is going to be installed in our docker container we need to finalize the installation. First we want to copy our custom made nginx configuration to the nginx folder in our docker container.

Then we want to copy our static folder, which includes our custom index.html, to our docker container. We copy the content of the static folder to a folder called “Publishers”.

Illustration 53: Finalize the installation

That’s it, we now have created our docker file. In the next chapter is explained how to build and run the docker container.

8.4 Running the web server as docker container

To run the Nginx web server in a docker container, we first need to build the container. This is done by opening a command prompt and entering the following command: “docker build -t nginx-modsecurity -f Dockerfile .” as shown in illustration 55. Don’t forget the dot at the end!

```
docker build -t nginx-modsecurity -f Dockerfile .
```

Illustration 54: Docker build command

In the terminal you can see all the steps (commands) we defined in the Dockerfile.
First we pull the base image, shown in the illustration below.

```
tep 1/7 : FROM owasp/modsecurity:3.0-nginx
.0-nginx: Pulling from owasp/modsecurity
```

Then we set the arguments, shown in illustration 56.

```
tep 2/7 : ARG BRANCH=v3.3/dev
---> Running in c9a77eacf327

Step 3/7 : ARG COMMIT=v3.3/dev
 ---> Running in ac07cbb7cac1

Step 4/7 : ARG REPO=SpiderLabs/owasp-modsecurity-crs
 ---> Running in e4f48ba7f35e
```

Illustration 55: Set arguments

Then we install the ModSecurity CRS, shown in illustration 56.

```
Step 5/7 : RUN apt-get update &&   apt-get -y install python git ca-certificates iproute2 &&   mkdir /opt/owasp-modsecurity-crs-3
in ${BRANCH} &&   git checkout ${COMMIT} &&  mv crs-setup.conf.example crs-setup.conf &&   ln -sv /opt/owasp-modsecurity-crs-3.2
/*.conf" >> /etc/modsecurity.d/include.conf &&   sed -i -e 's/SecRuleEngine DetectionOnly/SecRuleEngine On/g' /etc/modsecurity.d/
 ---> Running in edd2273befc2
```

Illustration 56: Install CRS

Then we finalize the installation by copying the static folder to the "/data/Publishers" folder in the docker container, as shown in illustration 57.

```
Step 6/7 : COPY /static /data/Publishers
 ---> 897675c3a086
```

Illustration 57: Copy static folder to container

The output of the command should look similar to the output shown in illustration 58.

```
Successfully built a81c18c67c83
Successfully tagged nginx-modsecurity:latest
```

Illustration 58: Build success

Now that we have successfully build the image, we want to run it. This is done by entering the command: "docker run -ti --rm -p 80:80 -e PARANOIA=1 nginx-modsecurity", as shown in illustration .

```
docker run -ti --rm -p 80:80 -e PARANOIA=1 nginx-modsecurity
```

Illustration 59: Run nginx-modsecurity

Some explanation for this command is as follows:

- **-ti** indicates that we want do not want to run the docker container in the background, now we can see the log's of the docker container.
- -**p** indicates the ports on which the container will be running. The 80 on the left of the :, means that the container is available on port 80 on the host. The 80 on the right of the :, means that the container is available on port 80 in the docker container.
- **-rm** makes sure the docker container is removed once it stops.
- - **e** PARANOIA=1, is used to set the degree of security. This ranges from 1 to 4, with 4 being the highest security level. This parameter is not required to run the command, because it's a default value.

The output of this command should be similar to the one shown in illustration. This line means ModSecurity is activated and the CRS is loaded.

```
2019/10/18 10:22:24 [notice] 1#1: ModSecurity-nginx v1.0.0 (rules loaded inline/local/remote: 0/896/0)
```

Illustration 60: Rule blocking dot dot slash attack

8.4.1 Validation

Now that we got the docker container up and running, we want to check if everything is working accordingly. First we want to navigate to http://localhost/. We will be greeted with our custom index.html we created in chapter 8.1, as shown in illustration 61.

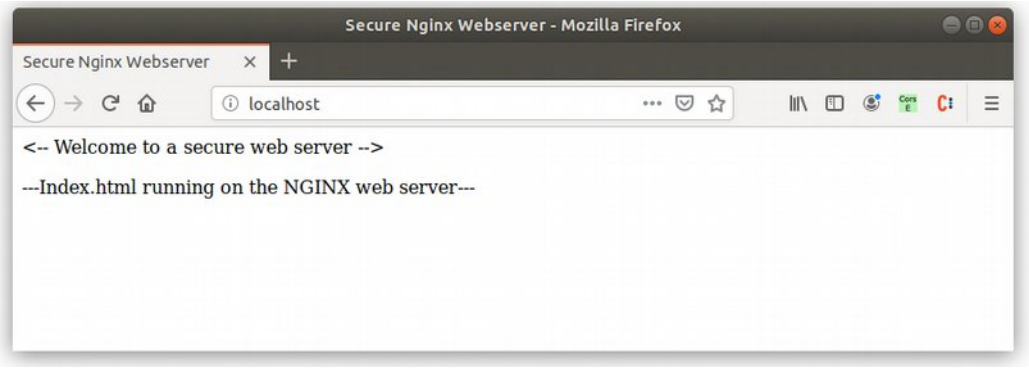


Illustration 61: Landing page

Now we want to confirm the rule set I loaded correctly. This can be done by navigating to : http://localhost/?abc=../../. If everything is setup correctly the error page, in illustration 62, will be shown.

Illustration 62: Error page

If you look at the command prompt now, you will see something similar to the output shown in illustration 63.

```
2019/10/18 10:44:03 [error] 6#6: *1 [client 172.17.0.1] ModSecurity: Ac
cess denied with code 403 (phase 2). Matched "Operator `Ge' with parame
ter `5' against variable `TX:ANOMALY_SCORE' (Value: `10' ) [file "/etc/
modsecurity.d/owasp-crs/rules/REQUEST-949-BLOCKING-EVALUATION.conf"] [l
ine "79"] [id "949110"] [rev ""] [msg "Inbound Anomaly Score Exceeded (
Total Score: 10)"] [data ""] [severity "2"] [ver ""] [maturity "0"] [ac
curacy "0"] [tag "application-multi"] [tag "language-multi"] [tag "plat
form-multi"] [tag "attack-generic"] [hostname "172.17.0.1"] [uri "/"] [
unique_id "157139544363.907129"] [ref ""], client: 172.17.0.1, server:
, request: "GET /?abc=../../ HTTP/1.1", host: "localhost"
172.17.0.1 - - [18/Oct/2019:10:44:03 +0000] "GET /?abc=../../ HTTP/1.1"
 403 154 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:69.0) Gecko/20
100101 Firefox/69.0"
```

Illustration 63: Error message in terminal

Some explanation for the output is as follows:
In illustration 64, you can see the rule that is related to the “attack” we executed, when navigating to http://localhost/?abc=../../.

```
[file "/etc/modsecurity.d/owasp-crs/rules/REQUEST-949-BLOCKING-EVALUATION.conf"]
```

Illustration 64: Rule to block dot dot slash attack

The attack we executed is known as the dot-dot-slash attack. When executing this attack we try to traverse the path to a desired path, with malicious intent. We now know the core rule set is successfully installed, since the web server blocked our simple attack.
In the next chapter I am going to show you how to get the flask web application running on the NGINX web server.

9 Connecting the pieces

At this point we have 2 separate containers. The two containers are as follows:

1) A flask-gunicorn container, with our flask web application
2) A nginx-modsecurity container, with our Nginx web server and our ModSecurity installed.

Now we have to make sure the Flask web application Is running on our secure Nginx web server. This is done by configuring Nginx web server to redirect us to the URL on which our flask web application is running. To make this possible we have to set an upstream server in our Nginx configuration file (nginx.conf).

9.1 Changing the NGINX configuration file: nginx.conf

There are 2 things we need to change in our Nginx configuration file. First we need to define the upstream server. An upstream server can be seen as an proxy. This is done inside the HTTP section of the Nginx configuration file, below the line where we set the location of the rules configuration file. How this is done, is shown in illustration 65.

```
# The code below is like assigning a value to a variable.
# we assign the server on which our flask application is running
# to a variable called app.
upstream app {
    # Set the upstream server ip and port
    # The ip and port we need to set are those
    # of the flask application:172.18.0.2:8000
    # We can write flask_app instead of 172.18.0.2
    server flask_app:8000;
}
```

Illustration 65: Setting upstream server

Now that we have set the upstream server we need to make sure it is passed when navigating to http://localhost/. How this is done, is shown in illustration 66.

```
server {
    # The server needs to listen on port 8000 because
    # flask application is running on port 8000
    listen       8000;

    # When navigating to localhost/ the upstream server is passed:
    # localhost:80/ --> 172.18.0.2:8000/index.html
    location / {
        proxy_pass http://app; #this actually means http://172.18.0.2:8000
    }
}
```

Illustration 66: Assign upstream server

That's it! We now have configured the Nginx web server,so that it will pass the Flask web application when navigating to http://localhost:80/.

In the next section we are going to set up the docker-compose.yml file.

9.2 Setting up the docker-compose.yml

The last thing we need to do is creating our docker-compose.yml file. This file makes sure we can easily run both containers at once, with one command.
So let's open our docker-compose.yml. We start with defining the version number of our docker-compose configuration. How this is done, is shown in illustration 67.

```
version: '3.7'
```

Illustration 67: version

Now that we have defined our version number, we can start creating the services of our docker-compose.yml. Add the line shown in the illustration below to the docker-compose.yml file.

```
services:
```

We are going to start off with creating our flask-gunicorn service. This is done in the service section we added in the previous step. How this is done, is shown in illustration 68.

- The name of the service is used to reference the IP address of the docker container.
- The name of the upstream server in the Nginx configuration file must match the service name in the docker compose file (flask_app in this case).

```
# Set the name of the service
flask_app:

  # Set the container name
  container_name: flask_app

  # Restart when container goes down
  restart: always

  # Set build location. This is
  # where the dockerfile is located
  build: ./flask-gunicorn

  # Set the ports of the container
  # If one is set it's the container port
  # If two are set it's:
  # {HOST}:{CONTAINER}
  ports:
  - "8000"

  # Set the command wich runs on startup
  command: gunicorn -b :8000 app:app

  # Set volume of the container.
  # This volume is the downloads folder in our
  # flask-gunicrion folder.
  volumes:
  - .:/downloads
```

Illustration 68: docker-compose Flask service

Now that we have created our flask service, we can start with the creation of the nginx-modsecurity service. How this is done, is shown in illustration 69.

```
nginx_modsec:

  # Set the container name
  container_name: nginx_modsec

  # Restart when container goes down
  restart: always

  # Set build location. This is
  # where the dockerfile is located
  build: ./nginx-modsecurity

  # Set the ports of the container
  # {HOST}:{CONTAINER}
  ports:
    - "80:8000"

  # This container depends on flask_app
  # This means that flask_app is started first
  # This means that flask_app is stopped last
  depends_on:
    - flask_app
```

Illustration 69: docker-compose Nginx service

That's it! We now have finished our docker-compose.yml. To run the docker-compose file, you have to navigate to the main directory of the project (20191017_Secure_Webserver_NGINX in this case) and run the command shown in illustration 70.

```
docker-compose up
```

Illustration 70: docker up

10 Bibliography

1) Yousrie, M. (2017, 3 december). *What is flask framework used for in Python? How is it different from Django and why would you prefer it over Django? - Quora*. Visited 16 October 2019, at https://www.quora.com/What-is-flask-framework-used-for-in-Python-How-is-it-different-from-Django-and-why-would-you-prefer-it-over-Django

2) Mackay, M. (2016). *WSGI Servers*. Visited 16 October 2019, at https://www.fullstackpython.com/wsgi-servers.html

3) Kasireddy, P. (2016, 4 maart). *A Beginner-Friendly Introduction to Containers, VMs and Docker*. Visited 16 October 2019, at https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/

4) NanoDano, . (2018, 10 lente). Nginx Tutorial. Visited 16 October 2019, at https://www.devdungeon.com/content/nginx-tutorial

5) Trustwace SpiderLabs. (z.d.). *ModSecurity Overview/What Can ModSecurity Do?* Visited 16 October 2019, at https://modsecurity.org/about.html

6) Trustwace SpiderLabs. (z.d.-b). *OWASP ModSecurity Core Rule Set (CRS)*. Visited 17 October 2019, at https://modsecurity.org/crs/