Politecnico di Torino

POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Degree Thesis

Container-based secure home gateway

Supervisor
prof. Antonio Lioy

Francesco Maria Folzi

Academic year 2021-2022

Summary

In a society where Internet has become a fundamental means of communication, the devices that enable the access to this immense network have assumed a fundamental role. Gateways allowing connection to the Internet are usually supplied by Internet service providers and offer a limited range of services. From a security point of view, these devices, along with other low-cost products on the market, often have configuration shortcomings that make them vulnerable to network attacks. They also do not provide users with the possibility to configure or implement other desired Internet services. This document proposes the development of an application that integrates a router, a switch, an access point, a firewall, an intrusion detection system (IDS) and a secure web browser. The application offers a high level of customization of services as well as the possibility of adding new functionalities. Software development is based on the popular container technology, allowing the isolation of individual services and increasing the level of security and portability on different devices. Container management is realized through the Docker container engine. The thesis deals with security issues in the container environment, analyzing the vulnerabilities and risks of this technology and suggesting solutions and practices to mitigate them. This overview is followed by the development of the application, starting with the explanation and containerization of the individual services and ending with their integration in a single network in the final product.

Contents

1 Introduction 7

2 Analysis on Container Security 10

2.1 Virtualization and Containers . . . 10

2.1.1 Containers . . . 10

2.1.2 Container images . . . 11

2.2 Docker overview . . . 11

2.2.1 Dockerfile . . . 12

2.3 Other container engines . . . 12

2.3.1 Podman . . . 13

2.4 Container orchestration . . . 13

2.4.1 Introduction . . . 13

2.4.2 Kubernetes . . . 13

2.4.3 Orchestrator risks . . . 14

2.4.4 Orchestrator countermeasures . . . 15

2.5 Container security . . . 15

2.5.1 Introduction . . . 15

2.5.2 Threats and vulnerabilities overview . . . 15

2.5.3 Solutions to secure containers . . . 19

2.5.4 KubeLinter security solution for Kubernetes . . . 22

3 Containers on Linux 25

3.1 Containers on Linux embedded systems . . . 25

3.1.1 Challenges . . . 25

3.2 Docker and Linux networking . . . 26

3.2.1 Linux bridge . . . 28

3.2.2 Iptables . . . 29

3.2.3 Docker network explained . . . 30

4 Project Components 31
4.1 Introduction . . . 31
4.2 Router with OpenWrt . . . 31
4.2.1 OpenWrt inside a container . . . 32
4.2.2 OpenWrt firewall . . . 37
4.3 Web browser with Firefox . . . 41
4.3.1 X11 Window System . . . 41
4.3.2 Firefox container . . . 42
4.4 IDS with Snort . . . 45
4.4.1 Snort rules . . . 45
4.4.2 Snort in a container . . . 46
4.4.3 Snort logs . . . 47
4.5 Switch and internal Firewall . . . 48
4.5.1 Linux bridges and iptables . . . 48
4.5.2 Switch and Firewall in a container . . . 48
4.6 Access Point . . . 52
4.6.1 Access Point inside a container . . . 52

5 Final project 59
5.1 Introduction . . . 59
5.1.1 Topology . . . 59
5.1.2 Programming language . . . 59
5.1.3 Setting the application firewall rules . . . 59
5.1.4 Configuration variables . . . 60
5.1.5 Host connection . . . 62
5.2 Application . . . 62

6 Experimental evaluation 63
6.1 Internet speed comparison . . . 63
6.2 Images total size . . . 64
6.3 Resources evaluation at runtime . . . 65
6.4 Throughput performance testing . . . 67
6.5 Detect port scans with Snort . . . 69

7 User manual 71
7.1 Overview . . . 71
7.2 Requirements . . . 71
7.2.1 System requirements . . . 71
7.2.2 Software requirements . . . 71
7.3 Configuration . . . 72

7.3.1 Change Wi-Fi name and password . . . 72
7.3.2 Change OpenWrt container password for SSH . . . 72
7.3.3 Change wireless card interface name . . . 73
7.3.4 Change Ethernet card interface name . . . 73
7.4 Build the images . . . 74
7.5 Application startup . . . 74
7.6 Terminate the application . . . 75
7.7 Application error . . . 75
7.8 Application usage . . . 75
7.8.1 Access the OpenWrt router container . . . 76
7.8.2 Manage the container services . . . 76
7.8.3 Connect devices to the Wi-Fi . . . 76
7.8.4 Use the Firefox web browser inside the container . . . 76
7.8.5 Enable and use the IDS . . . 77
7.9 Update the images . . . 77

8 Developer manual 78
8.1 Introduction . . . 78
8.2 Images and software . . . 79
8.3 Application components . . . 79
8.4 Application scripts . . . 80
8.4.1 Functions and APIs . . . 80
8.4.2 Build script . . . 80
8.4.3 Start script . . . 81
8.4.4 Stop script . . . 86
8.4.5 Connect host script . . . 86
8.5 Network details . . . 87
8.5.1 Router . . . 87
8.5.2 Switch . . . 87
8.6 Images configuration . . . 87
8.6.1 Internal firewall configuration file . . . 87
8.6.2 OpenWrt perimeter firewall configuration file . . . 88
8.6.3 IDS configuration file . . . 88
8.6.4 Web browser container SSH configuration . . . 88
8.6.5 Network interfaces . . . 88
8.6.6 Access point white-list and black-list access . . . 89
8.6.7 Configure access point DHCP IP addresses range . . . 89
8.7 Add new containers . . . 89
8.8 Download OpenWrt packages . . . 90
8.8.1 OpenWrt GUI . . . 90

9 Conclusion 91

Bibliography 92

Chapter 1

Introduction

With the advent of the Internet age, now well advanced but still developing, a lot of hardware components came into everyday life. While people are focused on computers, tablets and phones they tend to overlook and take for granted those that instead allow them to stay in touch with the entire web infrastructure: gateways. This term identifies a wide range of devices that allow connection between different networks. This document refers to the kind of gateway called residential or home gateway that allows a local area network (LAN) to be connected to a wide area network (WAN). This connection allows the hosts of a local area network access to the Internet. Gateways can have different configurations and integrate different functions, e.g. router, NAT, firewall, WiFi access point, switch. All these components have a specific function and may or may not be present on the device depending on the configuration. Routers, for example, allow connections between networks and control the data traffic between them. All the data and information sent through the Internet (e-mails, files, web pages) travels in the form of data packets.

When a customer that could be a person as well as a company needs to establish a connection to the Internet it contacts an ISP. An ISP (Internet Service Provider) is an organization that provides services for accessing and using the Internet. Typically the ISP with which customers stipulate a contract for the Internet connection also provides the home gateway that manages it. This is a common option that removes all the problems for customers such as buying the equipment and setting it up on their own. Moreover in case of malfunctions the ISP would provide assistance and solve the problem without any technical intervention of the customer. This could be a convenient choice for a home gateway of a customer that only needs Internet connection and is not concerned by other technical details. Unfortunately, contrary to what one may think, these kinds of devices are usually vulnerable to network attacks, expose security risks and don't offer a good flexibility in terms of services and functionalities. The same applies to other low-cost products on the market that can be purchased by users. For a customer or even a company concerned about these issues and who wants to implement network security and Internet services in a completely customizable way these devices are not a suitable choice.

Based on this prologue this document addresses one possible solution to the problem described. The thesis proposes the development of an application that integrates a router, a switch, an access point, two firewalls, an intrusion detection system (IDS) and a secure web browser. The application offers a high level of service customization as well as the possibility of adding new functionalities independently from the ISP. The application is compatible with any computer, device or embedded system in general that meets certain requirements. All the software used are free and for the most part open source. The application is developed with the trending technology of so called "container", which brings several benefits in terms of security and manageability of the application.

In recent times there has been a new trend in the development of applications towards the use of "microservices". This term refers to a new architectural style for the creation of applications different from the traditional monolithic approach. The microservices architecture consists in the division of the application in services of small size independent from each other. These services communicate with each other through special well defined APIs and provide the basic functionalities of the application.

In monolithic applications all the processes are strictly connected and are executed as a single service. This means, for example, that if the application workload is too high, the whole architecture needs to be resized. Adding a new feature to the application is complex because it needs to be well integrated in the base code. Also since the numerous processes are dependent from each other and tightly interconnected the malfunction of a single process can have a big impact on the entire application. These limits and defects are overcome by adopting a microservices architectural style. With the division in different, independent, little base services the application becomes more scalable, flexible and resilient. Each service executes and provides a single base function and is autonomous, which means that it can be developed, shared, modified without affecting the other components. Also an error of one service is isolated from the others and can be managed without blocking the entire application. Moreover from an organization point of view this approach proposes a restructuring of the development teams. Each different team becomes specialized in one particular service, making the development cycle faster and more independent. Obviously this is not the perfect architecture for every application and with all these benefits also come drawbacks like the complexity of the communication between services or the management of multiple database transactions. Anyway the microservices architecture fits well with the application addressed in this document.

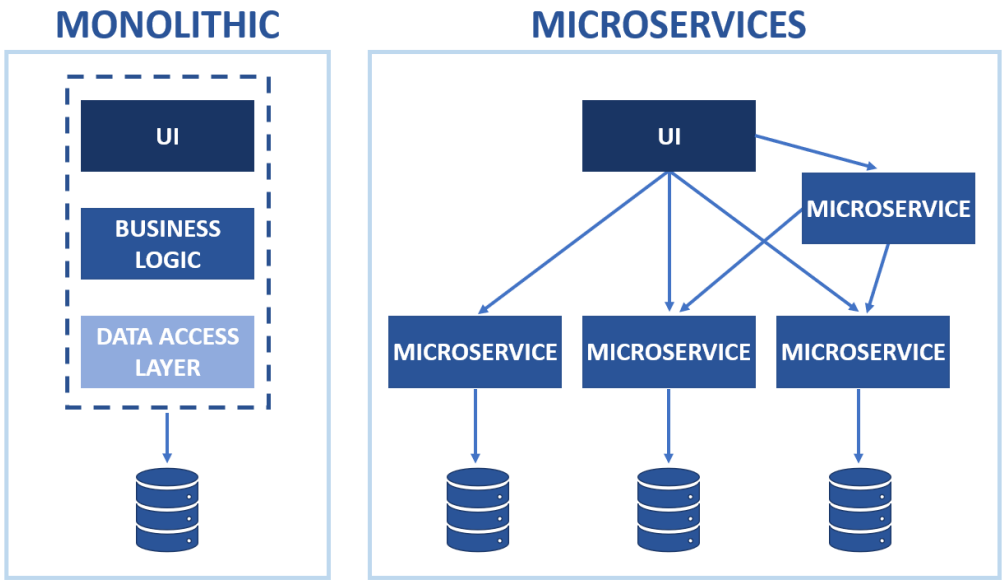

Figure 1.1. Virtual Machine and Container Deployments

Now that the architectural style has been explained, how can it be implemented on a practical level? Today's applications based on microservices are implemented using "containers". Containers are a lightweight form of virtualization that allow to package, distribute and run applications in different environments. This trend has had a great diffusion in recent years and it's used by the largest IT companies in various sectors (cloud native applications, normal applications, data centers). This technology allows to enclose in an image file everything needed (code, dependencies, libraries, configuration files) for the operation of a given service or application: it therefore makes it independent from the underlying hardware and software platform, and can be shipped and implemented in other software environments. Containers are isolated and independent from each other and take advantage of a lighter form of virtualization than traditional virtual machines allowing a quicker and more manageable deployment for applications. In an application based on microservices each of them can be included in a container appropriately connected to other containers, building an application which is composed by different isolated processes, manageable, shareable and fast to deploy in every environment compatible with this technology. There are different software for the management of containers, one of the most famous and popular is Docker, which is an open source project that manages Linux based containers through a client-server architecture.

The implementation of the application proposed in this thesis takes advantage of this technology born and refined in recent times and through Docker can be developed with a high level of security, manageability and segmentation where every process is isolated in a container with the appropriate connections.

This thesis focuses on the aspects concerning the implementation of containers, the creation of the network through which the containers exchange data and the analysis of the security associated with these processes. The first part of the document contains a general overview on those subjects whereas the second part concentrates on the application development. In the next chapter a more detailed explanation on containers will be given with a particular focus on the security aspects.

Chapter 2

Analysis on Container Security

2.1 Virtualization and Containers

Virtualization has been a very important subject during the last years. It applies to numerous fields of information technology, among which cloud computing that requires many instances of an application or software to run isolated one from the other but together on the same physical server. Lately a new form of virtualization has evolved in order to solve this kind of situation: application virtualization. Application virtualization differs in many ways from the older hardware virtualization and offers a lot of benefits and advantages in specific environments, such the one addressed in this document. Application virtualization offers a light, portable and reusable way to pack up applications and run them on different platforms. This kind of virtualization is realized with containers.

2.1.1 Containers

A container is a lightweight form of OS virtualization that allows to isolate applications from the underlying system. Containers provide a portable, quick and compact way to package and deploy applications reliably in different environments. One container can run anything from a single process to an entire application and can be created, started, stopped or deleted in any moment. Unlike traditional virtual machines that need to emulate all the system including the hardware and, on top of that, the entire OS, containers share the same underlying hardware and OS Kernel exploiting a virtualization only at application level as shown in figure 2.1. The main benefit that derives is a lighter, quicker and more manageable way of deploying apps in different environments. Containers that need to share information can be connected in networks. Typically complex applications are divided into smaller components, each one with its different specific task and running in its own container. These tasks are also called microservices.

All the containers on the same machine share the same OS kernel. Each of them runs a process or an application, is isolated from the others and has its own private file system and network stack. The most common operating systems used for containers are based on Linux. In Linux the container isolation is achieved by means of two Linux Kernel features: namespaces and control groups (cgroups).

namespaces are a feature of the Linux Kernel that allow the system to manage and isolate resources like Processes, Mount points, Inter Process Communications (IPC), Network stacks and Users. This is fundamental in a containerized environment since it allows every process running inside a container to have visibility only on its resources and not the ones of other containers.

cgroups are another Linux Kernel module that takes care of the access to hardware resources, limiting and managing every process access to memory, CPU, and disk;

To run a container it is necessary to have an image. A container is a running instance of an image.

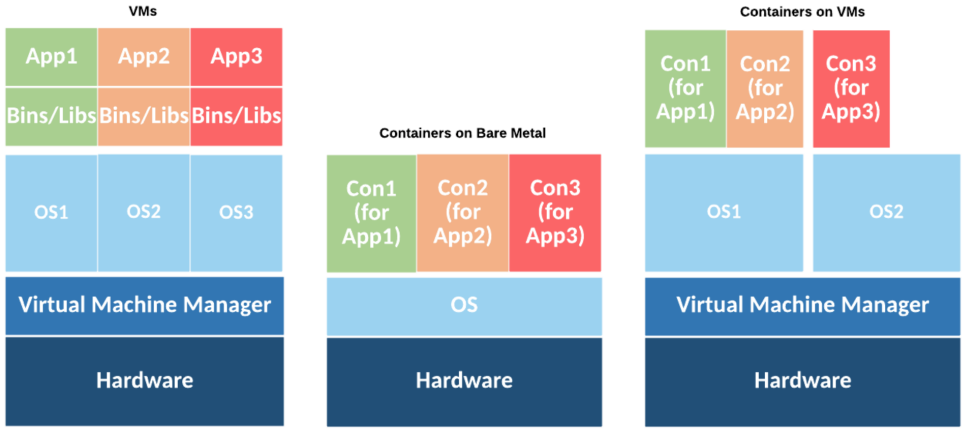

Figure 2.1. Virtual Machine and Container Deployments [1]

2.1.2 Container images

It is important to make a little overview on images since they are strictly related to containers and play an important role in the whole container lifecycle. An image is just a static template that contains all the instructions and files needed to run a container. Multiple equal containers can be started from the same image. It comprehends the libraries, tools and executable code required by the specific process that the related container should execute. Images make use of techniques like copy-on-write and layering to improve efficiency and decrease space occupation. An image is often divided in layers, the first layer (base layer) is typically an image of a common operating system for example Ubuntu or Alpine Linux. Then on top of that base layer the user can add more layers with specific programs, custom code or frameworks to build a unique image that satisfies his needs. This layering technique takes advantage of layer caching to speed up the process of building similar images. It follows that when an image is modified and rebuilt only the modified layers are rebuilt: that makes images lightweight and fast to build. It is possible to publish images in a public repository called Registry to facilitate sharing between users and developers.

2.2 Docker overview

This shift towards container technology necessitated the creation of tools to manage containers in a direct and simple way, with a high level of abstraction. Nowadays there are multiple choices of container engine but the most used is Docker. Docker is an open source project which allows the creation, shipping and execution of applications by means of Linux containers. It's written in Go programming language. This platform allows a fast and easy way to transfer and deploy containerized applications in every stage of their development and production in different environments (laptops, servers, cloud providers, embedded systems). Docker is the name of the technology as well as the name of the community and the actual company that works to improve and provide this service.

Docker is a containerization technology born on top of LXC functionalities which uses the already mentioned namespaces and cgroups Linux kernel features to isolate containers. Then in 2014 starting from version 0.9 Docker dropped LXC and introduced its own libcontainer library written using Go programming language. Libcontainer is currently the default container execution environment that allows the manipulation of control groups, namespaces, capabilities, network

interfaces and Apparmor profiles in a predictable way without depending on LXC. Anyway it's still possible to switch back to LXC every moment.

Docker implements a client-server REST architecture composed by the Docker client, the Docker daemon and Docker registries.

The Docker daemon is a Linux daemon that listens for Docker API requests by the clients and manages all the Docker components like images and containers. It is the core of the Docker system that builds containers starting from images and requires root access on the system.

The Docker client is a simple interface connected to the Docker daemon that calls Docker API based on a REST client-server architecture. It allows the user to directly interact with the Docker daemon through a Command Line Interface (CLI).

Docker registries are archives of Docker images where it is possible for organizations or single users to push and pull images in specific repositories. The default registry is Docker Hub which has public open-source images repositories as well as space where users can create their private repositories.

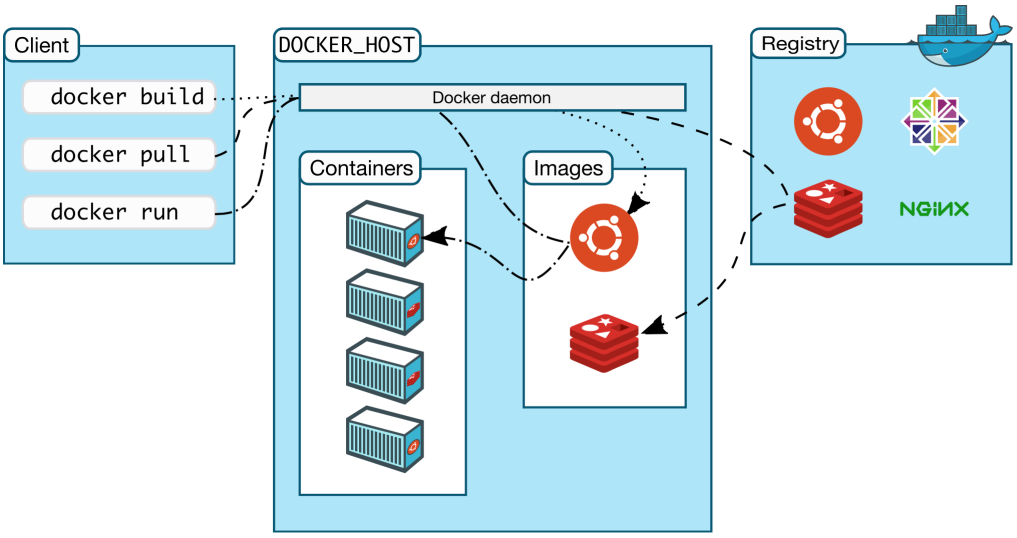

Figure 2.2. Docker architecture [3]

2.2.1 Dockerfile

A container is a running instance of an image. To build a Docker image it is necessary a so-called Dockerfile. A Dockerfile is a text file that contains all the instructions and commands to assemble an image. Using the command `docker build` users can create images starting from Dockerfile. This kind of file is going to be used multiple times throughout this document to create custom images.

2.3 Other container engines

Docker has changed the way applications are developed and maintained making containerization very popular but it's not the only option on the market. In fact, Linux containers are defined in a standard way and Docker is just one implementation to use them. The most interesting alternative is Podman which claims to be more secure and more lightweight than Docker.

2.3.1 Podman

Podman is an open source container engine written in Go and available on most Linux platforms. It's designed to build, share and run applications through containers. Podman was released in 2018 and was originally developed by Red Hat. This software is similar to Docker in many ways and uses almost the same CLI commands. This allows users to perform on Podman the same commands of Docker, facilitating the adoption of this software.

Despite the similarities there are two major differences that distinguish Podman from Docker. The first one regards the daemon. Docker relies on a daemon that runs in the background and manages all the Docker API calls from the Docker client and performs operations on containers. Podman instead is daemonless and all the commands executed through the CLI interact directly with the registry, the containers and the images storage. The second one is that, like Docker, Podman relies on Linux namespaces, which provide isolation for the processes, but contrary to Docker it does not require root privileges to run containers. Podman can run containers either as root or as an unprivileged user. It manages the entire ecosystem of containers, images and volumes through a library called "libpod". Another feature is the use of pods, which are simply the union of several containers inside the same Linux namespace, sharing the same resources.

There are two major benefits over Docker deriving from their differences. The first is that Podman doesn't rely on a single point of failure like the Docker daemon which is running in the background and managing all the processes. The second one is that containers can be created, run, and managed by users without administrator (root) privileges. Since the Docker daemon always runs with root privileges, in case an attacker manages to gain access to it, he could perform every sort of privileged operation with the containers on that host. However this scenario is improbable and brought to attention only for illustrative purposes.

2.4 Container orchestration

2.4.1 Introduction

Now that the idea of container has been explained, the resulting advantages and the tools have been introduced, it's necessary to bring the attention to what is called an orchestrator. Container orchestration is the automation of container deployment, management, scalability and networking processes. It is a highly functional methodology for companies that need to deploy and manage a really high number of containers for complex applications on multiple systems. Container orchestration is applicable to any environment where containers are used, and is useful for deploying the same application multiple times across multiple systems, without having to redesign it. Through automation the orchestration of containers helps to simplify the management of multiple operations such as provisioning and deployment of applications, resource allocation, monitoring containers health and scalability based on the workload across the infrastructure. Different orchestrators has been created to manage containers architecture scalability. The most popular are Kubernetes and Docker Swarm.

2.4.2 Kubernetes

Kubernetes is an open-source, portable platform that helps in the orchestration, configuration and automation of containerized applications, managing their workloads and services. Kubernetes was originally developed by Google and open-sourced in 2014. It's often abbreviated in K8s.

Containers help in packaging, shipping and running applications. In a production environment organizations need to manage a high number of containers and ensure that the applications work correctly with no downtime. In the case a container unexpectedly brokes, Kubernetes takes care of starting another container. Kubernetes provides a framework to run a resilient distributed system with containers, in particular it provides:

- Load balancing: if the traffic to a single node is too high, it automatically performs a workload balancing across the network to other nodes;
- Automated storage mounting system;
- Automated containers management: Kubernetes automatically scale containers based on the traffic or the organizations policy, it can deploy a new container or remove one and manage the released resources;
- Automated resources distribution: provided Kubernetes with a number of nodes and the resources (CPU and memory) each container should use, it will automatically deploys containers on the nodes in the best way;
- Container healing: containers are restarted, replaced or removed if they are failing or not responding;
- Secret storage: Kubernetes provides secure storage of sensitive information such as private keys or passwords.

Kubernetes introduces the concept of pods which is a group of several containers, with shared network resources, storage and rules to run the containers. The pods are "logical hosts" that run a containerized application in a cloud environment and can be compared to physical hosts that instead run the application on a physical machine.

2.4.3 Orchestrator risks

When analyzing container risks and threats also orchestrators need to be taken into consideration. Since the orchestrator plays an important role in the container environment and lifecycle, it's fundamental to analyze and limit the risks and vulnerabilities that an attacker could potentially exploit, possibly leading him to take control over the entire container ecosystem.

In the past many orchestrators were designed to let every user access as administrator. Each user could then have access to the entire containerized environment. However nowadays many orchestrators run different apps with different sensitivity levels managed by different teams. In these cases it's good practice to limit the access of every user only to its specific needs, avoiding involuntary or malicious user access to components that do not concern him.

In a multi-nodes containerized environment the traffic between hosts is realized through a virtual overlay network typically managed by the orchestrator. Traditional network management tools are not suitable for this kind of network because they would only see encrypted packets in transit between two hosts with no visibility of the real traffic being sent. Although the overlay network offers good security through encryption it creates a difficulty in monitoring the traffic in transit. The situation is even more critical if the different components are not well separated by sensitivity level allowing an attacker that has compromised for example a web site to use the network to attack other sensitive components.

The orchestrator manages containers balancing and scaling the workload by creating, moving or destroying a container when needed across different hosts based on the available resources. By default the orchestrator makes no difference between containers with different sensitivity levels, placing them possibly on the same host. This means that a critical vulnerability in a low-sensitivity container could open a breach to a high-sensitivity one on the same host.

It's fundamental to have a secure orchestrator configuration. A weak configuration could expose all the environment to a high risk. An example is that all the cluster of hosts should be well separated to avoid that the compromise of a single host results in the compromise of the entire cluster.

2.4.4 Orchestrator countermeasures

The orchestrator should be configured with a policy of least privilege access in which every user can access only the components, images, containers his role requires.

Orchestrator administrative accounts access should be monitored and controlled because these accounts grant access to all the resources of the environment. Strong authentication techniques must be adopted such as multi-factor authentication.

Network traffic should be separated in different virtual networks based on sensitivity level. For example the front-end of an application accessible by the public should use a virtual network while the back-end should use another one and the communication between front-end and back-end should be realized through a few well defined, monitored interfaces.

The deployment of containers in different hosts should be performed based on sensitivity level. The orchestrator should not place both high and low sensitivity workloads on the same host. Some rules should be defined to avoid this mix and separate the workloads on different hosts. Orchestrators provide methods to group containers together to achieve this separation. This allows a secure separation that decreases the attack surface and increases the layers of defense considering, for example, an attacker that is able to exploit a vulnerable public-facing container.

The orchestrator should be configured to ensure a secure environment for all the applications. All the nodes and their connectivity states should be monitored and logged. Orchestrator configurations and policies should ensure that the compromise of a single node does not affect the overall security of other nodes in the cluster. The tampered node should be isolated from the cluster without invalidating the other nodes or cluster operations.

2.5 Container security

2.5.1 Introduction

Lately containerized environments are becoming more and more used due to their speed, lightness and ease in managing applications in various systems. Obviously it is impossible that a change of this magnitude brings only benefits. Every modification in the technology adopted brings also new security aspects to take into consideration. Containers are intrinsically quite secure thanks to the concept of isolation. When evaluating container security, there are many important aspects to consider: image and container vulnerabilities, network connection, hardware resources and many others. This section addresses these aspects analyzing the risks and threats introduced by this technology and trying to give a solution for them. It is important to point out that containers do not intervene in the application security. This means that if the application that is going to be built in a containerized environment already exposes security problems then those vulnerabilities would still be present and an attacker can take advantage of them.

2.5.2 Threats and vulnerabilities overview

This section analyzes one by one the container vulnerabilities and threats. The image 2.3 represents a visual schema of the points addressed.

Kernel Exploits

Differently from a Virtual Machine, in a containerized environment each container shares the same underlying host kernel enlarging the attack surface. Therefore performing a kernel exploit inside a running container will have a direct impact on the host kernel. This means that if an attacker is able to exploit the host kernel then the isolation and security of the containers won't be guaranteed anymore because he could be able to take control over the entire machine. On a virtual machine instead this attack is more difficult because, before actually attacking the host kernel, the attacker should penetrate the VM kernel and the hypervisor.

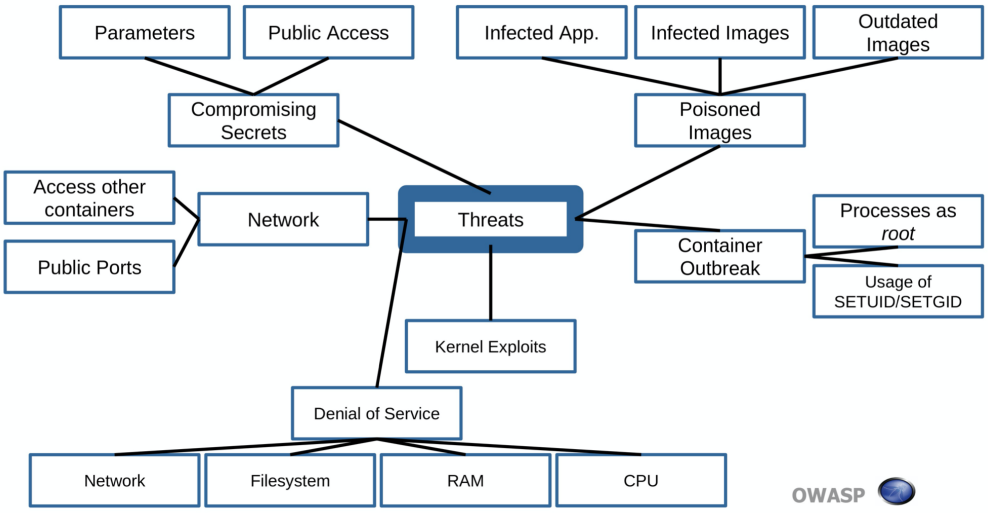

Figure 2.3. Docker threats [4]

For example, a common kernel privilege escalation exploit such as Dirty COW (CVE-2016-5195) performed inside a container leads to root access on the host machine.

There are multiple aspects to take into consideration to mitigate this problem. It's important to keep the host OS always updated to the latest version. Moreover there are "minimalistic" OS versions specially created for container services that reduce the overall attack surface like CoreOS, RancherOS, Project Atomic or Snappy Ubuntu Core. Also it's a good practice to apply the Mandatory Access Control (MAC) which is a kernel-level control that constrains the ability of a subject (user, thread, process) to perform various operations on an object or target of the system itself. AppArmor, Seccomp, SELinux are tools designed to perform this kind of control.

Here some best practices to mitigate kernel exploits:

- Keep the base system updated;
- Use minimal, container-specific host systems;
- Implement Mandatory Access Control to avoid undesired container operations at kernel level (Seccomp, AppArmor, SELinux).

Images risks

Images are fundamental in this kind of environment because they include all the components to run containers. Consequently any kind of error or misconfiguration present in an image will propagate to the relative running container. There are different aspects to analyze to determine that an image is secure and won't expose any vulnerability.

Since any component of an image like OSs or programs will run in a container it's necessary to analyze them first. Those components may be outdated or not updated to the latest version and therefore present some security issues or vulnerabilities that may be exploited when running the container. For example a web app running inside a container may be vulnerable to a XSS attack or similarly a database could be exploited through an SQL injection. Eventually a compromised container could be used to damage or steal sensitive data from other containers or the host itself. It's important to always keep all the components updated to the latest secure version. This needs to be done continuously because new vulnerabilities may be discovered at any time.

Since containers are running instances of images, to update a program it's necessary to change the image itself and then redeploy the container. Another aspect to take into account is the image configuration, for example an image may be configured to run with a specific user account. If

this part is missing then the container will run with a privileged user and thus create unnecessary risks exposing the machine to a privilege escalation attack. Another misconfiguration could be an unnecessary SSH daemon included in the image that exposes the container to useless network risks.

Usually images are built starting from other images taken from registries. It's fundamental to check the integrity of the base image and to download it from a trusted source. Solutions such as PKI (Public Key Infrastructure) or hash digest are used to check the authentication and the integrity of an image. Docker offers its own trusted registry called Docker Hub. Using images from third parties whose reliability cannot be verified can be dangerous as they can include malwares or software with known vulnerabilities. When starting the container such malwares own the same capabilities of the other components in the image, leading to a possible attack to other running containers or even to the host machine.

Another important aspect is that applications often need secrets like passwords or keys to enable secure channels of communication, access services or perform sensitive operations in general. Though it is possible to embed those data into the image itself this could be a risk because anyone that can access the image can parse it and steal the secrets. It is therefore best practice to store them in another secure place and retrieve them at runtime.

Here some best practices to mitigate images risks:

- Images should be downloaded from trusted sources (e.g. Docker registry Docker Hub);
- Use Docker content trust feature to verify the integrity of the image and the authenticity of the publisher;
- Periodically update and rebuild images to apply new security patches;
- Implement a security scanner like Clair.

Container outbreak

Container outbreak means that a container escaped its isolation managing to gain access to other containers, to the host machine or acquiring new privileges. A common configuration mistake is to let a container run as root where it is not necessary. If an attacker is able to exploit the service inside that container and end up in a fully privileged container then he could be able to access the host machine and steal sensitive information or perform damaging operations. For example one risk is that a container with high privileges may be allowed to mount sensitive directories exposing the host filesystem to a risk of file tampering or data leak. To avoid this possibility is good practice to set filesystems and volumes to read-only where possible. Every container should run with a least privilege policy and there are multiple ways to achieve this.

There are different possibilities to run a container as a non-privileged user. If someone is creating his custom image the recommendation is to create directly in the Dockerfile a user with a chosen uid and run the service with that user. Otherwise if the image is pulled from a registry it is possible to rebuild another image starting from that one as the base layer and then adding the user as before. In any case it's always possible to specify the user when starting a container. However, doing it directly in the image makes the container secure by default.

Another way of running a container as a non-privileged user is to create an isolated user namespace on the host. A process's user ID can be different inside and outside that namespace. In particular, a process can have a normal unprivileged user outside the namespace while at the same time having a root user inside it. Considering a container that runs as "fake" root in that namespace it can only perform full privileged operations that are limited to that namespace.

Another aspect to take into consideration is Linux capabilities. Linux kernel capabilities are a set of privileges that can be given to any process to grant it permission to perform intended privileged tasks. Those privileges were normally reserved for root processes (ID 0). Now Linux divides the privileges traditionally associated with the root user into distinct units called capabilities, which can be independently enabled or disabled for every process. It's possible to drop

or add capabilities based on what privileges a container needs. Basically no process should own all the privileges but only those intended to perform its specific task. A good practice is to drop all the capabilities of a container and then add only those necessary. Also it's better to run the Docker containers with the `no-new-privileges` option to prevent privilege escalation attack through `setuid` binaries.

It's worth noting that a container started as an unprivileged user has by default no capabilities.

Here some best practices to avoid container outbreak:

- Drop unnecessary capabilities and enable `no-new-privileges` option;
- Monitor dangerous mount points on the host (e.g. `/var/run/docker.sock`, `/proc`, `/dev`);
- If possible give just read-only privileges on the file system;
- Don't run containers as root if it's not strictly necessary, run them as a non-privileged user.

Compromising Secrets

Any application that needs to access a service typically requires some secrets. A secret could be a password, an API key, an SSH key, a certificate or any kind of sensitive data which must not be disclosed. These secrets should not be put in a Dockerfile neither in the application source code. Learning the Docker architecture and using some best practices to secure this data will help to mitigate risks.

Here some best practices to avoid secrets disclosure:

- Don't store secrets in environment variable;
- Don't store secrets in the container image;
- Implement secure secrets storage.

Network

It's fundamental to correctly design the network architecture of the application. In a classic application network scenario where the application provides services to an external network, security is achieved thanks to a well designed DMZ (demilitarized zone) and components like firewall and IDS, making sure that only the front-end service is reachable from the Internet. The front-end is then securely connected to the back-end. An application network well segmented and correctly organized and monitored implements a good level of security. This is the same approach to adopt when planning a network in a containerized environment with microservices. With containers the risk is that a badly-designed network results in a flat, unsegmented and insecure network where any microservice can interact with the others. The complexity that derives from a container-based infrastructure is that since each microservice typically runs in a different container and some of them need to communicate to each other, it may be complicated to design a good network. And while some containers need to be connected, others should definitely not. If a container is compromised the more network connections it has, the larger the attack surface. For example it could be used to scan the network in order to find vulnerabilities or weaknesses in other containers connected to it.

Here are some general recommendation to build a multilayers well segmented network:

- Choose the network devices which best suit the application environment;
- If the application needs to expose services to the Internet realize the proper DMZ segmentation with firewall and IDS to guarantee protection and inspect the packets;
- Expose to the network only the ports that are strictly necessary;

- Implement only the connections between containers that need to communicate;
- Secure the management APIs, don't expose them to the Internet nor in the DMZ, implement whitelisting;
- Secure the host services;
- The communication with Docker registries should be TLS encrypted.

Resources protection

All the containers running on a host share the same Kernel and the same underlying hardware. This means that CPU, memory, disks I/O and network resources of the host are shared between all the active containers. Typically in these environments there are a large number of containers running concurrently racing to have access to all these resources. The risk is that either a bug in the software or an unexpected software behavior or a deliberate attack by some malicious user can lead to a denial of service in terms of hardware resources. To mitigate this problem it's possible to limit adequately the resources of CPU, memory, network, swapping and other kernel resources that any container can use. Docker offers all kinds of commands to manage and impose limits to those resources when starting containers. By default, a Docker container has no resource constraints and can use as much of a given resource as it needs. For this reason it is good practice to check that all containers are started with the required limitations.

Here some best practices to mitigate resources abuse:

- Use Docker commands options to limit the system resources accessible by containers;
- Test the workload in pre-production;
- Monitor the resources in use by containers.

Registry risks

A registry is an archive where images are stored by users or organizations. It can be either private or public. These images can be retrieved anytime and used to run containers. Images can contain sensitive data which is transmitted over the network when they are retrieved from the registry. If the channel over which the image is transmitted is insecure, an attacker could easily have access to those sensitive components. Moreover there is the risk of a man-in-the-middle attack to steal sensitive information or upload fraudulent images.

Another problem is outdated images. Since most of the images that are deployed are typically those taken from the registries, then an outdated or vulnerable version of an image results in a vulnerable container. All the images in the registries need to be monitored and kept updated.

Lastly an insecure access to a registry with insufficient authentication requirements could lead to an unauthorized access of an attacker that could then maliciously upload or download data from the registry.

2.5.3 Solutions to secure containers

Container technology has revolutionized the way applications are developed and deployed. Shipping, deploying and managing applications has never been so easy as now with containers. Unfortunately with new technologies there are also new security challenges that both security teams and developers must face carefully. Moreover being a fairly new technology some security issues are still in a phase of definition and analysis.

Now that the main vulnerabilities and weaknesses of containers have been shown with some short advice on how to mitigate them, this section proceeds to analyze in depth what are the solutions and best practices to build a secure containerized environment. Furthermore this section focuses on the way these practices can be included in the application development cycle from the point of view of an organization.

Images countermeasures

Old vulnerabilities management tools and processes don't integrate well with the new technology because of the different ways of running applications through images and containers. There is a need for new tools and processes to validate the effective security of images. These operations should reflect the pipelined-based build structure and immutability of images and containers to bring good results. Some key aspects need to be followed.

- A complete integration with the lifecycle of images, starting with the pull from the registry or the custom build to the actual use when running a container;
- Given the layered structure of images, an in-depth analysis of vulnerabilities should be executed on every layer, from the base layer to the custom ones used by the organization;
- At the end of every stage of the build process it's necessary to check that the images respect all the security policies of the organization. For example checking if there are vulnerabilities in the image with Common Vulnerability Scoring System (CVSS) that exceed a given threshold and in case stop the development of the image.

Some image vulnerability scanners have already been created and can be adopted in the security evaluation process.

A similar procedure should be done to check that all the images respect the configurations and settings best practice. For example, if not necessary, an image should not run as a privileged user (root) but as an unprivileged user that needs to be configured. SSH or other remote connection tools should never be active by default inside a container to avoid unnecessary network attacks and reduce the attack surface. Images compliance state must be continuously monitored and updated if needed.

Since images are static files that can contain malwares, continuous monitoring should be executed by the organization using malware signatures pattern and behavioral detection.

No kind of secret or sensitive data should be stored inside the images to avoid an attacker parsing the image and finding it. Instead the secret should be given during runtime to the specific container that requires it without storing it in the disk. The transit of secrets over the network must always be encrypted with cryptographic algorithms approved by the Federal Information Processing Standard (FIPS).

Organizations should always use trusted images, that are reliable images taken from trusted sources that don't expose containers to known vulnerabilities. The key aspects to mitigate this risks includes:

- A list of trusted images drawn up and updated from the organization to make sure that only the images on that list are used;
- Identification of each image by a NIST-validated cryptographic signature implementation;
- Before execution every image needs to be signature validated;
- Use only trusted repositories that are continuously monitored and maintained and don't use third-party untrusted images.

Containers countermeasures

Containers are active instances of images with actual processes running inside them. This means that any vulnerability could lead to an attacker exploiting it and breaking into the system causing potentially serious damages. For this reason organizations should use tools to scan containers for Common Vulnerabilities and Exposures (CVEs) and quickly intervene if any problem occurs.

Organizations should use tools to monitor and assess configuration settings in the container environment. These tools should frequently check if the configuration settings of the containers

respect the configuration standards and if not enforce them. For example, if it is not necessary, a container should not run as a privileged user (root) to prevent container outbreak attacks. Instead configure the container to run as an unprivileged user if possible.

Containers should be deployed with the strictly necessary file system permissions. Containers should not mount local file systems on the host but, if changes to files need to persist on the disk, specific storage volumes need to be allocated for this purpose. Organizations should use tools to enforce this policy and prevent violations.

Fundamental is to enable mandatory access control (MAC) technologies such as AppArmor and SELinux. MAC is a type of OS access control that limits the access to resources based on sensitivity level. It constrains the ability of a subject (e.g. user) to perform various operations on an object or target of the system itself. This technology mitigates the possibility of a compromised container to access sensitive resources by adding a new level of security and segmentation. MAC technology should be used from organizations to ensure that containers have access to specific paths, files and processes that they need. AppArmor and SELinux are two Linux security modules that implement this technology, the difference is that the first is easier to manage and works with paths while the second is more difficult to maintain and is based on labels.

Another Linux security module is Seccomp (Secure computing) that allows limiting the system calls from the containers to the kernel. Custom actions can be specified when a certain system call is executed. Container system calls are restricted to those strictly needed to reduce the attack surface. Docker for example includes a default Seccomp profile to drop unsafe system calls, but it's alway possible to create custom profiles to best meet the requirements.

Old intrusion detection tools for monolithic applications are unable to detect in container-based apps due to the different technology. Organizations should use new container aware tools that well integrate with the new environment. The ideal is to use tools that automatically build security profiles for containers based on behavioral learning. These profiles should detect and mitigate anomalies such as unexpected processes, system calls, writings and changes to files, network traffic and malwares storage or execution.

All the containers should be associated with individual identities and logged to supply a full trace of activity and prevent rogue containers.

Host countermeasures

To reduce the attack surface it is recommended to use a container-specific OS where possible. These kinds of minimalistic OSs are specifically designed to work with containers. They already implement security features such as read-only file systems and other hardening practices. For those organizations that cannot use such specific OSs but general purpose ones, it is highly recommended to follow the NIST SP 800-123, Guide to General Server Security [2] to minimize the attack surface. For example an host with running containers should not run any other apps or services outside a container. Finally every layer of the host machine, from kernel to containers, should be continuously scanned for updates and vulnerabilities.

No data or states or application dependencies should be persistently stored on the host. Instead all these components should be packaged into containers, providing a host operating in a stateless and immutable manner with a minimum attack surface and making it easier to identify lower level anomalies.

Organizations should make sure that login and authentication to hosts is audited and logged to ease the identification of anomalous access patterns and unexpected privileged operations.

The host filesystem should be set as read-only to prevent file tampering, if possible.

Registry countermeasures

The connection from hosts to registries should be always performed over secure encrypted channels. All the data pulled or pushed to the registries should be encrypted in transit.

For what concerns outdated images there are two solutions. Organizations can totally remove old, outdated or vulnerable images from the registry when they are no longer needed. Another way is to use different tags to specify the image version, for example application:1.2, application:2.3. In this way developers always know which version they are deploying.

Finally the access to registries must require secure authentication techniques so that only trusted images are uploaded. A good practice is to let developers upload images only to repositories that concern them instead of giving them access to all the repositories.

All the write and read accesses to a registry should be audited and logged. Moreover organizations should use tools to scan images for vulnerabilities and perform a compliance assessment before pushing them to the repository.

Network best practices

The complexity of building a secure network derives from the containers virtual isolation. Each container should perform its own service and be connected to other containers in order to exchange data. The best solution is to build a performing secure segmented network where each container is connected to others based on the service it provides and its sensitivity level, limiting the connections to those necessary and avoiding connecting containers that don't have to communicate.

Organizations should use tools to monitor the traffic between containers and at network borders, ensuring that only the necessary ports are open, that the traffic is legit and that containers are not connected to networks of differing sensitivity levels. For example containers that host secure data should not be connected directly to the Internet.

Limitation of capabilities

Capabilities are a feature of the Linux kernel that provides a set of privileges that can be given to processes and containers. There is a list of privileges, from the permissions to change the system time to the permission to mount a filesystem or even enable kernel auditing. By default container engines such as Docker already provide containers with a minimum set of capabilities that can be customized where required. An even more secure solution is to remove all capabilities from a container and then enable only those strictly necessary. An attacker that successfully penetrates into the container won't be able to easily break out of the container because the actions he can perform on the host and kernel are limited.

Limitation of hardware resources

By default a container has access to all hardware resources such as CPU, GPU and memory without any limitation. If an attacker successfully breaks into a container he could perform a DoS attack, saturating the resources and blocking other services. Containers should be limited on the amount of resources that they can use. Docker provides a list of command options to limit container resources.

2.5.4 KubeLinter security solution for Kubernetes

KubeLinter is an open-source command line tool to identify misconfigurations in Kubernetes YAML files and Helm charts. KubeLinter runs multiple checks for security errors and DevOps best practices. Some misconfigurations identified are for example running containers as root or containers that expose port 22 with SSH. KubeLinter is configurable, checks can be enabled or disabled and it's possible to create custom checks. KubeLinter does not automatically solve the issues but, when a check fails, gives recommendations on how to patch it. Since this tool is in an early stage of development, at the moment KubeLinter provides around 40 security checks but anybody can join the community and contribute to its development.

Here is an example of the usage of KubeLinter taken from the official documentation site [7], the box below shows a sample pod file `pod.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

This sample file has two production readiness issues and one security issue.

Security issue

- The container in this pod is not running as a read-only file system, allowing it to write to the root filesystem.

Production readiness issues

- The configuration doesn't specify the container's CPU limits, allowing it to consume excessive CPU;
- The configuration doesn't specify the container's memory limits, allowing it to consume excessive memory.

To check this file with KubeLinter, run the following command:

```
kube-linter lint pod.yaml
```

KubeLinter runs the default checks and reports errors.

```
pod.yaml: (object: <no namespace>/security-context-demo /v1, Kind=Pod)
    container "sec-ctx-demo" does not have a read-only root file system
    (check: no-read-only-root-fs, remediation: Set readOnlyRootFilesystem to
    true in your container's securityContext.)

pod.yaml: (object: <no namespace>/security-context-demo /v1, Kind=Pod)
    container "sec-ctx-demo" has cpu limit 0 (check: unset-cpu-requirements,
    remediation: Set your container's CPU requests and limits depending on
    its requirements. See https://kubernetes.io/docs/concepts/configuration/
```

```
     manage-resources-containers/requests-and-limits for more details.)

pod.yaml: (object: <no namespace>/security-context-demo /v1, Kind=Pod)
    container "sec-ctx-demo" has memory limit 0 (check:
    unset-memory-requirements, remediation: Set your container's memory
    requests and limits depending on its requirements. See
    https://kubernetes.io/docs/concepts/configuration/
     manage-resources-containers/requests-and-limits for more details.)

Error: found 3 lint errors
```

Chapter 3

Containers on Linux

3.1 Containers on Linux embedded systems

Container technology has become increasingly popular in recent years in many fields, including embedded systems. A lot of operating systems and tools have been developed to help such integration like Torizon, BalenaOS or Linux microPlatform. These software provide a minimal Linux system equipped with a Docker infrastructure to manage, build and deploy images and containers in an easy way.

Creating a containerized environment on an embedded system is a difficult challenge due to its intrinsic characteristics. Several aspects need to be taken into consideration like the limited hardware resources that require building a lightweight and minimal system. Another example could be the network access, which not all the embedded systems include. Fortunately a well designed container ecosystem fits well with some of these characteristics.

The advantages that containers bring are numerous. They allow to develop an application without caring about the underlying platform. The application environment is independent from the base operating system and there are not incompatible dependencies. The application can be executed on different platforms. The development cycle of the base OS and the one of the application are decoupled, meaning that the upgrading of the two can be done in different times when needed.

Other benefits brought from containerization are:

- container isolation;
- decouple the embedded OS from the application development;
- low overhead;
- faster development;
- limited hardware integration.

3.1.1 Challenges

Despite all the benefits that help in the integration of containers and embedded systems, there are also some relevant challenges to consider in the process.

Hardware resources limitation

Embedded systems are special purpose, they are designed to offer a specific service. Based on this they are built as low consumption systems with limited hardware resources, such as disk storage, CPU and RAM. The containerization needs to be realized based on these potential constraints. Containers should use as minimum resources as possible and this is not always easy, for example when running a fully distro-based container.

Data integrity and reliability

One of the major risks for an embedded device is data corruption. If the root filesystem is compromised wrong behavior may occur. Therefore safety countermeasures should be adopted to prevent and detect any kind of corruption. A good practice is to configure the filesystem as read-only so that any write operation that would tamper with it is denied.

Cross-platform development

There are a multitude of devices with different characteristics, different CPUs or architectures. Although containers can be shipped and deployed in different environments these aspects need to be taken into account when building a containerized application. For example an application built for a x86 platform and deployed on a platform with a different architecture like an ARM based one won't work correctly. Handling cross-platform development is necessary to make the best out of different systems.

System provisioning

Some embedded devices may not have an Internet connection and this affects the design and build of the application. A system with no network connection should have all the components already present in it. For a containerized environment all the container images could be for example deployed in the firmware image so that they are ready to be used.

System update

In a containerized system the application and the base OS could have different development cycles. This also means different times for the update. The update time and method should be decided before the development.

Container traceability

Docker containers traceability is a challenge as the deployment process may sometimes break. Finding all the containers history, modifications and dependencies as well as the build and deployment stages can save a lot of time. A performing traceability system is a mandatory requirement for today's industrial world.

License compliance

Software licenses are always a challenge during application development. Each container image inherits the legal obligations for each file and software. Legal risks should be minimized by paying attention to license requirements. It's important to maintain and track license compliance for each software.

3.2 Docker and Linux networking

To understand and realize the application addressed in this document it's necessary to have a deep knowledge of Docker networking. All the containers are going to be isolated and connected over a network. This section analyzes theoretically and practically how Docker network commands and tools take advantage of Linux features to implement the network.

Container engines virtualize network resources to isolate containers by means of network namespaces. Namespaces are an abstraction of global system resources which are visible only to the processes within the namespace. This resource abstraction permits to isolate instances of

resources and every different namespace sees only the resources assigned to it. There are 7 types of namespaces: IPC, Cgroup, Network, PID, Users, Mount, UTS.

With network namespaces container engines assign containers a dedicated network stack and each container can not interact with other containers' network resources. A process in a namespace has its own network interfaces, devices and stack.

The Linux command to create an isolated network namespace called "ns1" is

```
ip netns add ns1
```

and to run a process inside that namespace

```
ip netns exec ns1 [COMMAND]
```

for example

```
ip netns exec ns1 python3 -m http.server 8000
```

The command `ip netns exec <namespace-name> [COMMAND]` runs the process `[COMMAND]` inside the network namespace specified by name. The default namespace of the host is called `default` and every service or program executed on the host terminal will be assigned to that namespace. To list all the namespaces the command is `ip netns`.

Isolated namespaces can be connected with a Linux feature called veth. Veth stands for virtual Ethernet and is a network device that acts as a tunnel between network namespaces. It can be considered as a virtual Ethernet cable. A veth has two endpoints and can be created with the syntax

```
ip link add <p1-name> type veth peer name <p2-name>
```

In the code above "p1-name" and "p2-name" are the two names of the veth endpoints. Data transmitted in one of the two device endpoints will immediately be received on the other one. When one of them is down the link state of the pair is down. To move one endpoint in another namespace the command is

```
ip link set <p1-name> netns <namespace-name>
```

and to set an endpoint interfaces in up state:

```
ip link set <p1-name> up
```

Bringing all together as an example, here is reported a script to create and connect two isolated namespaces together through the default host namespace. Those are also the commands that Docker uses to build networks between containers.

```
#!/usr/bin/env bash

#first namespace variables
NS1="ns1"
#first namespace endpoint
VETH1="veth1"
#default namespace endpoint
VETHNS1="vethns1"

#second namespace variable (as above)
NS2="ns2"
VETH2="veth2"
VETHNS2="vethns2"

# create namespaces
ip netns add $NS1
ip netns add $NS2

# create veth devices
ip link add ${VETH1} type veth peer name ${VETHNS1}
```

```
ip link add ${VETH2} type veth peer name ${VETHNS2}

# setup veth interfaces up on the default namespace
ip link set ${VETH1} up
ip link set ${VETH2} up

# move veth end points into namespaces
ip link set ${VETHNS1} netns ${NS1}
ip link set ${VETHNS2} netns ${NS2}

# setup veth interfaces up on the ns1 and ns2 namespace
ip netns exec ${NS1} ip link set ${VETHNS1} up
ip netns exec ${NS2} ip link set ${VETHNS2} up
```

At this point both the namespaces are connected to the default one but have no network connection. In order to connect a namespace to the network it needs to have at least one interface and this interface must have an IP address assigned. The following commands assign an IP address to the veth endpoints in the ns1 and ns2 namespaces:

```
ADDRNS1="192.168.2.10"
ADDRNS2="192.168.2.20"

# assign ip address to ns interfaces
ip netns exec ${NS1} ip addr add ${ADDRNS1}/16 dev ${VETHNS1}
ip netns exec ${NS2} ip addr add ${ADDRNS2}/16 dev ${VETHNS2}
```

The interfaces inside the two namespaces have now an IP assigned. The interfaces on the host default namespace do not need an IP because they will be connected through a bridge.

3.2.1 Linux bridge

A network bridge is a device that connects two Ethernet segments together, it creates a single network connecting two or more network segments. The connection is protocol independent, which means that all protocols can pass through a bridge transparently. The packets are transmitted based on Ethernet address (MAC) and not IP address like routers do. While routers allow the communication between two separated networks, bridges merge the networks into a single enlarged one. Docker creates a bridge called "Docker0" as a default bridge for connecting containers. Containers can communicate with each other or send and receive packets to the Internet all through Docker0.

The following piece of code illustrates how to create a bridge, assign it an IP and connect the previous interfaces to it.

```
BR_ADDR="192.168.2.1"
BRIDGE="br0"

# setup bridge
ip link add ${BRIDGE} type bridge
ip link set ${BRIDGE} up

# assign veth pairs to bridge
ip link set ${VETH1} master ${BRIDGE}
ip link set ${VETH2} master ${BRIDGE}

# setup bridge ip
ip addr add ${BR_ADDR}/16 dev ${BRIDGE}
```

Now the two namespaces are connected to the bridge but they don't have the route tables updated to communicate. A default route need to be added in each of them to communicate, specifically the one of the bridge:

```
# add default routes for ns
ip netns exec ${NS1} ip route add default via ${BR_ADDR}
ip netns exec ${NS2} ip route add default via ${BR_ADDR}
```

The namespaces are now properly connected and ready to be tested, for example with a simple ping command:

```
ip netns exec ${NS1} ping ${ADDRNS2}
PING 192.168.2.20 (192.168.2.20) 56(84) bytes of data.
64 bytes from 192.168.2.20: icmp_seq=1 ttl=64 time=0.045 ms
64 bytes from 192.168.2.20: icmp_seq=2 ttl=64 time=0.039 ms

ip netns exec ${NS2} ping ${ADDRNS1}
PING 192.168.2.10 (192.168.2.10) 56(84) bytes of data.
64 bytes from 192.168.2.10: icmp_seq=1 ttl=64 time=0.045 ms
64 bytes from 192.168.2.10: icmp_seq=2 ttl=64 time=0.039 ms
```

The namespaces are correctly connected and visible to each other. The next step is to connect them to the Internet. For this purpose another Linux feature needs to be introduced: iptables.

3.2.2 Iptables

Iptables is a command line utility to configure the kernel-level firewall of Linux, which is based on netfilter modules. Iptables is based on a mechanism of RULES which determines if the incoming or outgoing packets need to be accepted or not. Each rule is part of a rules CHAIN. When a packet arrives, a specific chain is checked rule by rule to decide if it has to be accepted or dropped. There are 5 different chains: INPUT, OUTPUT, FORWARD, PREROUTING and POSTROUTING.

INPUT works on the packets entering the system right before been delivered to a local process;

OUTPUT works on the packets exiting the system after being created by a local process;

FORWARD works on the packets transiting on the system directed to another host;

PREROUTING works on packets right after being received by an interfaces, before INPUT;

POSTROUTING works on packets right before leaving an interfaces, after OUTPUT.

Each chain is associated with a TABLE. There are three tables: FILTER, the default table used for processing packets. NAT, used to change the source or destination address of packets. MANGLE, used to modify packets. The table/chain combination is the following:

- FILTER: Input, Output, Forward;
- NAT: Prerouting, Postrouting, Output;
- MANGLE: Prerouting, Postrouting, Input, Output, Forward.

Returning to the script, to allow the traffic of the namespaces to exit the host machine and reach the Internet it is necessary to create an iptables rule:

```
# enable ip forwarding in case it is not already enabled
bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'

iptables -t nat -A POSTROUTING -s ${BR_ADDR}/16 ! -o ${BRIDGE} -j MASQUERADE
```

This command adds a rule to the POSTROUTING chain into the NAT table. The command MASQUERADE allows to change the source IP address of the packets with the IP address of the outgoing interface. Now all the outgoing bridge traffic will be masqueraded and sent out.

This is how Docker manages all the network-related operations for containers underneath the Docker networking commands. Now that it's clear how Docker uses the tools and the features from Linux to build networks and connect containers, the next section analyzes the Docker network options.

3.2.3 Docker network explained

Docker creates an isolated network namespace and network stack for each container. It automatically assigns an IP, a default gateway, a routing table and DNS service to containers. It offers different options for the network.

Host networking

Host networking can be specified by adding the option `--net=host` in the Docker command `docker run [OPTIONS] IMAGE` when starting a container. In this configuration the container shares the same default network namespace of the host. Checking the network interfaces from within the container or from the host will show identical results.

Bridge networking

Bridge networking allows connecting containers to the default bridge Docker0, this can be done specifying `--net=bridge` when starting the container or by default without specifying the network at all. This creates an isolated network namespace for the container and connects it to the Docker0 bridge using a veth exactly as seen in the previous section. In fact, checking the network interfaces on the host, a new veth interface appears. Containers connected to the same bridge can communicate. The communications that go outside the host are masqueraded (NAT) as seen before.

Custom bridge networking

This configuration is similar to the one before except for the fact that the bridge is created by the user. The option to add when starting a container is `--net=<bridge-name>`. The custom bridge needs to be created before connecting any container to it. Docker offers networking commands to create custom bridges. When the bridge has been created it appears on the host network interfaces.

Macvlan

Macvlan is a lightweight driver that allows the user to assign a MAC address to a container exactly as if it was a physical device on the network. It does not make use of Linux bridge or port mapping, it connects the container interface directly to a specified host interface. The container is then given an IP address on the subnet of the external network connected to the host interface.

No networking

Specifying `--net=none` when running a container allows to create an isolated container with no network connection. Only the loopback interface is present.

Chapter 4

Project Components

4.1 Introduction

This chapter analyzes from a practical perspective the development of the project and explains the building process of the application. The application is developed and tested on a Ubuntu 20.04 LTS system with x86-64 architecture, then will eventually be published in order to be usable by users on different systems thanks to the portability that containers offer. The container engine adopted for managing images and containers is Docker. The application is divided into microservices which run individually or in pairs in an isolated container. These containers are securely connected over a custom network. First the microservices of the application need to be illustrated. The services that the system incorporates are:

- A router to connect the private internal network with the public external one;
- A switch to connect all the multiple internal services and hosts to the network;
- An IDS (intrusion detection system) to increase the security by detecting malicious traffic;
- A perimeter firewall to create a barrier between the internal and external networks;
- An internal firewall to monitor and manage internal network traffic;
- An access point to allow WI-FI connection to the network;
- A web browser to have a secure sandboxed web connection.

The services described run on separated containers offering a higher segmentation avoiding a single point of failure. The containers could be conveniently removed and restarted afresh if needed. The following sections analyze one by one the programs used to realize the services and the actual practical process for containerizing them as it was performed during development. After that the network topology aspect will be analyzed together with the final application implementation.

4.2 Router with OpenWrt

OpenWrt (open wireless router) is an open-source distribution based on Linux that target embedded devices. The project is specifically design for wireless gateways but can be used on other devices such as microcontrollers or personal computers. The purpose of this project is to extend the functionalities of common access gateways provided by ISPs. It is optimized and small enough to fit into system with limited resources. The software provides a rich archive of packages that can be installed through opkg, which is a full lightweight package manager for the root filesystem.

OpenWrt can be configured through a command line interface or more comfortably through a web interface called LuCi. The software is based on Linux, musl, util-linux and BusyBox.

The purpose of the project is to run OpenWrt inside a container, to take advantage of the container isolation, minimize the attack surface and avoid a single point of failure.

4.2.1 OpenWrt inside a container

To run OpenWrt inside a container it's necessary to have an image. The official images repository of the project can be found on Docker Hub under openwrtorg/rootfs. This repository offers several OpenWrt base images for different devices and CPU architectures. The following image is built for a system with x86-64 architecture but there is also an image with a different tag for a system with ARM architecture.

To pull the image from the repository:

```
sudo docker pull openwrtorg/rootfs:x86-64
```

then, to verify that the image has been pulled, the command

```
sudo docker images
```

print a list with all the images present on the host.

```
openwrtorg/rootfs x86-64 1a1dfba2b4e9 5 weeks ago 8.29MB
```

The image of OpenWrt is correctly downloaded and has a size of 8.29 MB. The next step is to understand how to connect the container to the external network. In a router perspective, the WAN (Wide Area Network) is the upstream network which means the existing LAN in case the router runs behind another router (for testing and developing purposes) or the Internet in case it runs as the main home router. In the case of a normal home router the hardware is directly connected to the WAN with an Ethernet cable. In the case of a container running a router it's necessary to reproduce this connection in order to have it directly connected to the WAN. This is possible by moving the Ethernet interface of the system directly into the container namespace. This operation is easy but brings a downside that needs to be considered: moving from the default system namespace the Ethernet interface that, being connected to the WAN, provides the Internet connection will lead to a lack of Internet connection in the main host (assuming that this Ethernet interface was the only Internet source). The container instead will have direct connection through the interface. This is not a problem since, if needed, the host can still connect to the Internet through the LAN created by the application, as will be shown later in the document.

The network interfaces can be listed with the command `ip a`:

```
user@user:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
    default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp6s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
    group default qlen 1000
    link/ether 74:d0:2b:46:c8:3b brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.6/24 brd 192.168.1.255 scope global dynamic noprefixroute
        enp6s0
       valid_lft 86308sec preferred_lft 86308sec
    inet6 fe80::758a:824b:13dd:211a/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: wlp7s0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN group default
    qlen 1000
```

```
    link/ether b8:76:3f:4e:43:25 brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
    DOWN group default
    link/ether 02:42:b0:f6:80:69 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::2248:53f5:f80:8f71/64 scope link
       valid_lft forever preferred_lft forever
```

The Ethernet interface connected to the WAN is "enp6s0" and will be moved to the container namespace after it is started. Note that the interfaces names are those of the system on which the application was developed and tested and may vary on other machines. To start the container:

```
sudo docker run --cap-add=NET_ADMIN -it --name=openwrt_1 --net=none
    openwrtorg/rootfs:x86-64
```

This command runs a container based on the openwrtorg/rootfs:x86-64 image with the name "openwrt_1" which is not yet connected to any network (`--net=none`). The flag `-it` is a short for `--interactive` and `--tty` and creates immediately an interactive shell in the container attached to the current terminal. The command `--cap-add=NET_ADMIN` gives to the container the Linux capability `CAP_NET_ADMIN` which allows it to perform various network related operations and is necessary since the container runs a router. After the command, if no error occurred, the container is started and the current terminal is directly connected to the command line interface of OpenWrt as shown in figure 4.1.

Figure 4.1. OpenWrt CLI

After starting the container the "enp6s0" interface can be moved to its namespace. Before this process there is a little modification that needs to be done on the host.

When Docker creates a container, it creates a new network namespace and links it to the path `/proc/<container-id>/ns/net`. Most of the code that is going to be used is based on the `netns` namespaces utility command. This command requires that the namespace files are in the default location `/var/run/netns/<container-name>`. This allow commands like `ip link set <interface-name> netns <container-name>` to work on the correct namespace. To solve this problem it's necessary to create a symlink of the containers namespaces from the first directory to the second one. The commands are:

```
sudo mkdir -p /var/run/netns/
c1pid=$(sudo docker inspect -f '{{.State.Pid}}' <container-name>)
sudo ln -sfT /proc/${c1pid}/ns/net /var/run/netns/<container-name>
```

The first command creates the directory in case it doesn't already exist. The second command extracts the container ID in a variable that is used in the third command to create the symlink. This procedure should be done for all the containers whose namespace needs to be manipulated with the `netns` command. At this point it's possible to move the "enp6s0" interface to the router container namespace and enable it with the commands

```
sudo mkdir -p /var/run/netns/
c1pid=$(sudo docker inspect -f '{{.State.Pid}}' openwrt_1)
sudo ln -sfT /proc/${c1pid}/ns/net /var/run/netns/openwrt_1

ip link set enp6s0 netns openwrt_1
ip netns exec openwrt_1 ip link set up enp6s0
```

The container with OpenWrt is connected and ready to be configured. The network configuration of a router typically consists of two interfaces called "wan" and "lan" which in this case are associated with the devices `enp6s0` and `br-lan`. Note that "lan" and "wan" are the name of the interfaces that implement the actual LAN and WAN. `enp6s0` is the Ethernet connection to the upstream WAN network moved to the container namespace. `br-lan` is the bridge to which all the hosts on the LAN should connect to get access to the Internet. The configuration of the router is managed by the UCI system. UCI stands for Unified Configuration Interface and is a command line program to centralize the configuration of the router. There are several files under the directory `/etc/config/` and each file relates to the part of the system it configures. Those files can be modified in different ways. They can be edited with a text editor or modified through the UCI command line program. The file to configure the network is `/etc/config/network` and on the running container has the following content (command `cat /etc/config/network`):

```
root@b63e3ba8196b:/# cat /etc/config/network
config interface 'loopback'
    option ifname 'lo'
    option proto 'static'
    option ipaddr '127.0.0.1'
    option netmask '255.0.0.0'

config interface 'wan'
    option ifname 'eth0'
    option proto 'dhcp'

config interface 'wan6'
    option ifname 'eth0'
    option proto 'dhcp6'
```

The default file shows that the "wan" interface is present and associated to the device `eth0` with the protocol DHCP for the IP management. `eth0` is the default and most common Ethernet interface name but isn't correct in the current situation. This file must be modified according to the application needs: the "wan" device must be changed to `enp6s0` and the "lan" interface which is not present needs to be created by editing the network file as follows:

```
root@b63e3ba8196b:/# cat /etc/config/network
config interface 'loopback'
    option ifname 'lo'
    option proto 'static'
    option ipaddr '127.0.0.1'
    option netmask '255.0.0.0'

config interface 'lan'
    option type 'bridge'
    option ifname 'veth2b'
```

```
    option proto 'static'
    option ipaddr '192.168.16.1'
    option netmask '255.255.255.0'
    option ip6assign '60'

config interface 'wan'
    option ifname 'enp6s0'
    option proto 'dhcp'

config interface 'wan6'
    option ifname 'enp6s0'
    option proto 'dhcp6'
```

Now the "lan" interface is created. It is associated with a device called `veth2b` which is a veth endpoint that will be connected to the switch container in the final application and will be explained better later in this document. The interface is of type bridge, which results in the creation of a bridge called `br-lan` with the `veth2b` device connected. The protocol for the IP assignment is `static` and the address is `192.168.16.1` with the specified netmask. The "lan" and "wan" interfaces must have different subnets for routing to work.

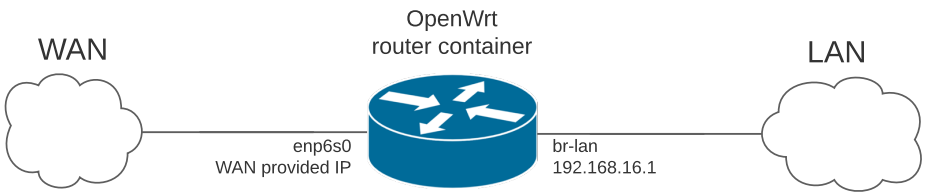

Figure 4.2. OpenWrt router Diagram

At this point the router in the container is properly configured but when the container is deleted and another OpenWrt container is started from the same base image the configuration changes made are lost. The way to fix this issue is to write a Dockerfile based on the OpenWrt base image, add the configuration file and then build the new image. The purpose of the Dockerfile is to build an image which has the network configuration already set up. To perform this operation it's necessary to create a folder containing two files: the Dockerfile and the network file with the "lan" interface and the "enp6s0" device as shown above. The Dockerfile should contain the code:

```
FROM openwrtorg/rootfs:x86-64
```

```
COPY network /etc/config/

CMD ["sbin/init"]
```

The command `FROM` specifies the base image to modify, `COPY` adds the specified network file in the `/etc/config/` folder as required. `CMD` specifies what command has to be executed on container start. At this point it's time to build the image with the command:

```
sudo docker build . -t router:1.0
```

This command builds a new image called "router" and tagged "1.0". Note the `.` which specifies the current directory to build the image, meaning that to run the command the current directory must be the one containing the Dockerfile and the network configuration file. Now the container can be started:

```
sudo docker run --cap-add=NET_ADMIN -it --name=openwrt_1 --net=none router:1.0
```

And the network configuration is already set up since the `/etc/config/network` file is now updated. In fact when the devices `enp6s0` and `veth2b` are moved into the container namespace OpenWrt detects them and enables the LAN. In fact showing all the network interfaces inside the container gives the following result:

```
root@552ab917ed15:/# ifconfig
br-lan    Link encap:Ethernet  HWaddr 12:BE:EA:1A:65:70
          inet addr:192.168.16.1  Bcast:192.168.16.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:450 (450.0 B)

enp6s0      Link encap:Ethernet  HWaddr 74:d0:2b:46:c8:3b
          inet addr:192.168.1.6  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

veth2b    Link encap:Ethernet  HWaddr 12:BE:EA:1A:65:70
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:34 errors:0 dropped:0 overruns:0 frame:0
          TX packets:41 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5721 (5.5 KiB)  TX bytes:6502 (6.3 KiB)
```

The `br-lan` device is the one specified in the network file for the LAN with IP `192.168.16.1` and `veth2b` attached. Any machine or other container connected to the `br-lan` with an IP in the LAN subnet and the default gateway set to the `br-lan` IP will be successfully connected to the newly created containerized router as part of the LAN.

OpenWrt SSH

There are two different methods of accessing the container. The first is to access it from the host with the command `sudo docker attach openwrt_1`. This attaches the terminal to a shell inside the container. The other method is through SSH.

OpenWrt listens for SSH connections on port 22 by default. It uses a software package called Dropbear that is designed to implement SSH (client and server) on devices with low memory and processor resources, for example embedded systems. Access is only available to devices within the LAN created by the router, with the command `ssh root@192.168.16.1`. By default when the container with OpenWrt is started there is no password set. To avoid security risks the password should be set immediately after starting the container. This process is automated in the final project of the thesis with a password chosen by the user.

The use of SSH within a container brings a disadvantage concerning host keys, which are automatically generated within the container. If these keys were automatically generated whenever the container is restarted, they would be different each time and devices that had already connected with SSH to the container by associating its old keys would no longer recognise the container and raise an error. In order to always start the container with the same keys, it is possible to store them in a folder on the host and mount that folder inside the container each time it is started. The folder that stores the host keys within the container is `/etc/dropbear/`. As this folder is already correctly configured in the previous active container, it can be copied to the host as a folder to be mounted on future OpenWrt containers. To perform this operation the following command can be executed from the host terminal:

```
sudo docker cp <containerId>:/etc/dropbear .
```

It is a Docker command that copies the "dropbear" folder of the container to the current directory on the host, which must be the one containing the Dockerfile. At this point, to start the container and mount the host keys folder inside it, the command is

```
sudo docker run --cap-add=NET_ADMIN -it --name=openwrt_1 --net=none
    $(pwd)/dropbear:/etc/dropbear:ro router:1.0
```

The option `-v $(pwd)/dropbear:/etc/dropbear:ro` mount the"dropbear" folder present in the current directory (`pwd`) inside the container at the path `/etc/dropbear` as read-only (`:ro`). This folder contains the host keys of the container which will be the same each time it is started, avoiding SSH connection issues. Note that the current terminal directory needs to be the one with the Dockerfile and the "dropbear" folder.

4.2.2 OpenWrt firewall

In addition to the router, OpenWrt offers many other services, including the implementation of a default perimeter firewall. This component increases the security level of the entire network creating a barrier between the internal private network (LAN) and the external public network of Internet (WAN) and for this reason it is used in the final project. This section explains the modules, rules and parameters of the default firewall.

The firewall takes care of filtering all traffic passing through the router. Packets can be dropped, rejected (dropped with an appropriate response to the source) or accepted. The OpenWrt firewall is based on the Linux Netfilter project, which is a default kernel framework in all Linux distributions that implements several network-related operations such as packet filtering or network address translation. Furthermore OpenWrt uses an application called fw4 (Firewall4) to provide the firewall. This application has different firewall provisioning mechanisms. The main configuration file is `/etc/config/firewall` and can be edited to change the firewall settings. The default content of the firewall is as follows:

```
config defaults
        option syn_flood 1
        option input  ACCEPT
        option output  ACCEPT
```

```
        option forward  REJECT
# Uncomment this line to disable ipv6 rules
# option disable_ipv6 1

config zone
        option name  lan
        list network  'lan'
        option input  ACCEPT
        option output  ACCEPT
        option forward  ACCEPT

config zone
        option name  wan
        list network  'wan'
        list network  'wan6'
        option input  REJECT
        option output  ACCEPT
        option forward  REJECT
        option masq  1
        option mtu_fix  1

config forwarding
        option src  lan
        option dest  wan

# We need to accept udp packets on port 68,
# see https://dev.openwrt.org/ticket/4108
config rule
        option name  Allow-DHCP-Renew
        option src  wan
        option proto  udp
        option dest_port 68
        option target  ACCEPT
        option family  ipv4

# Allow IPv4 ping
config rule
        option name  Allow-Ping
        option src  wan
        option proto  icmp
        option icmp_type echo-request
        option family  ipv4
        option target  ACCEPT

config rule
        option name  Allow-IGMP
        option src  wan
        option proto  igmp
        option family  ipv4
        option target  ACCEPT

# Allow DHCPv6 replies
# see https://dev.openwrt.org/ticket/10381
config rule
        option name  Allow-DHCPv6
        option src  wan
        option proto  udp
```

```
        option src_ip  fc00::/6
        option dest_ip  fc00::/6
        option dest_port 546
        option family  ipv6
        option target  ACCEPT

config rule
        option name  Allow-MLD
        option src  wan
        option proto  icmp
        option src_ip  fe80::/10
        list icmp_type  '130/0'
        list icmp_type  '131/0'
        list icmp_type  '132/0'
        list icmp_type  '143/0'
        option family  ipv6
        option target  ACCEPT

# Allow essential incoming IPv6 ICMP traffic
config rule
        option name  Allow-ICMPv6-Input
        option src  wan
        option proto icmp
        list icmp_type  echo-request
        list icmp_type  echo-reply
        list icmp_type  destination-unreachable
        list icmp_type  packet-too-big
        list icmp_type  time-exceeded
        list icmp_type  bad-header
        list icmp_type  unknown-header-type
        list icmp_type  router-solicitation
        list icmp_type  neighbour-solicitation
        list icmp_type  router-advertisement
        list icmp_type  neighbour-advertisement
        option limit  1000/sec
        option family  ipv6
        option target  ACCEPT

# Allow essential forwarded IPv6 ICMP traffic
config rule
        option name  Allow-ICMPv6-Forward
        option src  wan
        option dest  *
        option proto  icmp
        list icmp_type  echo-request
        list icmp_type  echo-reply
        list icmp_type  destination-unreachable
        list icmp_type  packet-too-big
        list icmp_type  time-exceeded
        list icmp_type  bad-header
        list icmp_type  unknown-header-type
        option limit  1000/sec
        option family  ipv6
        option target  ACCEPT

config rule
        option name  Allow-IPSec-ESP
```

```
        option src  wan
        option dest  lan
        option proto  esp
        option target  ACCEPT

config rule
        option name  Allow-ISAKMP
        option src  wan
        option dest  lan
        option dest_port 500
        option proto  udp
        option target  ACCEPT
```

All the configuration above is implemented in the default firewall. A minimum configuration for a router usually consists of a "default" section, two "zones" (`lan` and `wan`) and a "forwarding" section to allow traffic from `lan` to `wan`. The following paragraphs present the section types that may be used in the firewall configuration.

Defaults

The `default` section implements the global firewall settings that do not belong to specific zones. In the configuration file above this section sets the global settings for the INPUT, OUTPUT and FORWARD chains. Moreover the line `option syn_flood 1` enables SYN FLOOD protection by default.

Zones

The `zone` section groups one or more interfaces and is used in other sections as source or destination parameter. The firewall configuration file contains two zones, `lan` and `wan`, associated with their respective interfaces and sets policies for the chains INPUT (packets arriving at an interface in the zone), OUTPUT (packets leaving an interface in the zone) and FORWARD (packets transiting the zone). Moreover the `wan` zone also implements MASQUERADING with the option `option masq 1`. IP masquerading is a NAT method that allows a local network using private addresses to connect to the Internet via the public address of a router, in this case the one of the `wan` interface.

Forwardings

The `forwarding` section allows the traffic flow from a source `zone` to a destination one. In the configuration file the section

```
config forwarding
        option src  lan
        option dest  wan
```

allows the traffic flow from `lan` to `wan`.

Rules

A `rule` section defines a single rule that specifies if packets to/from specific ports or hosts should be accepted, dropped or rejected. Options of this section include source zone (`src`), destination zone (`dest`), IP address, MAC address, protocol, port, target and allow the creation of specific rules to control traffic. If both source and destination options are specified, the rule matches the forwarded traffic. If only one option between source or destination is specified, the rule matches respectively incoming or outgoing traffic. If neither of the two options is specified, the rule matches the outgoing traffic.

The rules used in the default firewall configuration file are explained in short comments preceding the rules and allow the following operations and relative traffic:

- Accept udp packets on port 68 to allow DHCP renew;
- Allow IPv4 ping;
- Allow DHCPv6 replies;
- Allow essential incoming IPv6 ICMP traffic;
- Allow essential forwarded IPv6 ICMP traffic;

All the IPv6 rules can be disabled if needed by uncommenting the line

```
# option disable_ipv6 1
```

at the beginning of the file. The last two rules allow IPsec/ESP and ISAKMP passthrough and are default rules for IPv6 gateways defined by IETF organization (for more information refer to https://datatracker.ietf.org/doc/html/rfc6092).

4.3 Web browser with Firefox

This section explains theoretically and practically how to build a containerized web browser and how to use it. The web browser chosen for this project is Firefox. It is a common browser that satisfies all the requirements needed. It will run inside an isolated container and this brings lots of advantages for what concerns the security of the system, reducing the application attack surface. For example in case of an erroneous download of a malicious file from the web browser, it would be isolated in the container, without affecting and damaging the host. The container could then be easily destroyed and restarted with a new unaffected web browser.

4.3.1 X11 Window System

Since containers do not offer a graphical interface, the web browser GUI is going to be forwarded using the X11 Window system also known as X11 or X, which is an architecture-independent system for remote graphical user interfaces used in Unix-like environments. X provides a simple way to access the GUI of a program from any display connected to the system. It could be the display of the host or the display of a remote machine. It provides a basic interaction with a mouse and a keyboard as well as moving and resizing the GUI window.

X uses a client-server architecture. The clients are the programs running remotely or on the same machine. The server is the host connected to those programs displaying the GUIs. The common way to access a X client application is through SSH. The steps to perform are:

- On the X server machine open a terminal;
- Connect to the X client remote machine with SSH specifying the Y forwarding argument in the command;
- Request local display service.

These steps will be better analyzed in the following part.

4.3.2 Firefox container

To build the image it's necessary to start from a base image. It could be any base OS image such as Alpine or Ubuntu. Alpine is a good choice given his lightness. Then in the Dockerfile the base image needs to be modified by adding the required packages, in this case all the packages to run Firefox, SSH and X11.

First the Dockerfile is created inside a new folder and edited as follows.

```
FROM alpine

RUN apk update && apk add --no-cache openssh openrc firefox xauth
    libcanberra-gtk3 ttf-opensans

CMD["/bin/sh"]
```

The command `RUN` executes any commands in a new layer on top of the base image and commits the results. In this case the installation of software packages through the alpine packets manager apk. The software installed are:

- openssh: the tool for remote login using SSH protocol;
- openrc: an init system for Unix-like computer based on dependencies, it helps in the management of system services;
- firefox: the web browser itself;
- xauth: the program used to manage the authorization information used when connecting to the X server;
- ttf-opensans: a packet required for the GUI characters visibility through X;
- libcanberra-gtk3: a package required by Firefox to run over X.

The `CMD` command specifies the entry point command to be executed when starting the container, in this case a simple bash shell. Now the Dockerfile is ready, to build the image from inside the directory:

```
sudo docker build . -t firefoxalpine:1.0
```

If no error occurs the image should be created and added to the local images archive with the name and tag "firefoxalpine:1.0".

At this point the container can be ran with the command:

```
sudo docker run --name firefox -it firefoxalpine:1.0
```

Without the network specification the container is connected to the default Docker bridge `Docker0`. At this point the container is running and has Firefox installed. In order to activate X applications through SSH some more configurations need to be done. Specifically the "sshd" service which is the OpenSSH server process needs to be activated, listening for incoming SSH connections. To accomplish this first the system needs a password. The command to add a password is `passwd` which asks to type a password two times. The password is for root since the system is running as root. Then some changes need to be made to the file that configures the SSH server: `/etc/ssh/sshd_config`. The changes are:

- The line PermitRootLogin should be uncommented and set to yes. This allows the SSH connection as root authenticating with the only password. This brings security risks and is done just for demonstration purpose, other methods for authentication should be used;
- The line X11Forwarding should be uncommented and set to yes;

- The line X11DisplayOffset 10 should be uncommented;
- The line X11UseLocalhost should be uncommented and set to no.

Another thing to check is that the hostname of the container is associated with its IP in the file `/etc/hosts`. This avoids name resolution problems when forwarding applications with X11. If the line is missing it needs to be added. At this point some configuration commands need to be executed to enable the services. The commands `rc-status` and `touch /run/openrc/softlevel` configure the correct operation of OperRC. Then the sshd service can be started:

```
service sshd start
```

If the system does not yet have the host keys for SSH, as in this case, they are generated automatically. Host keys are cryptographic keys used for authenticating computers in the SSH protocol. They are generated in pairs using RSA, DSA and ECDSA algorithms and stored in the container at the path `/etc/ssh/`, together with the configuration files.

To SSH into the container from the host or a remote system (every machine that can reach the container over a network can connect to it) the command is:

```
ssh -Y root@<ip_of_container>
```

The `Y` flag enable the use of X11 during the connection, `root` means that the connection into the container is performed as root and `<ip_of_container>` is the IP of the host, in this case the container which the connection is directed to. This IP can be found by simply running `ip a` inside the container or by using the command `docker inspect firefox` on the host to list the container informations. The password asked to access the container over SSH is the password of the root account into the container. Once the password is inserted, if all went correctly, a shell into the container should appear. At this point there is an active SSH connection to the container. Since X11 is enabled, the GUIs of the applications started on the container through SSH should be forwarded to the remote machine that created the connection. For example running the command `firefox` should open a GUI of the firefox application running inside the container.

In the figure 4.3 it can be noticed that the Firefox application runs inside the container by looking at the top bar writing "Mozilla Firefox (on bc936cf30c32)", which is the hostname of the container.

It's possible to add a layer of security by enabling the SSH service on a non privileged user rather than SSH as root. The reason is that the root account has full privileged access to the system and an attacker that successfully penetrates into the network and gets a shell in the container would have full privileged control over it. In order to avoid this eventuality the solution is to create a non privileged user that cannot run dangerous commands or access sensitive files. The command to create it inside the container is `adduser <username>`, then a password is requested for the new user. Also in the file `/etc/ssh/sshd_config` the line PermitRootLogin can be set to "no" to totally disable the SSH connection as root. The command to SSH into the container as the new user becomes:

```
ssh -Y <username>@<ip_of_container>
```

All this configuration has been performed in the active container just to demonstrate how to enable all the services. If the container is deleted and restarted from the image the configuration is lost. To avoid this problem it's necessary to incorporate the setup within the Docker image by changing the Dockerfile and building a new image with the right configuration. The only problem is the host keys automatically generated within the container. If these keys were automatically generated whenever the container is restarted, they would be different each time and devices that had already connected with SSH to the container by associating its old keys would no longer recognise the container and raise an error. In order to always start the container with the same keys, it is possible to store them in a folder on the host and mount that folder inside the container each time it is started. Considering the container manually configured in the previous paragraphs, the folder concerned is `/etc/ssh` which contains all the host keys and configuration files. As this folder is already correctly configured, it can be copied to the host as a folder to be mounted on future containers. To perform this operation the following command can be executed from the host terminal:

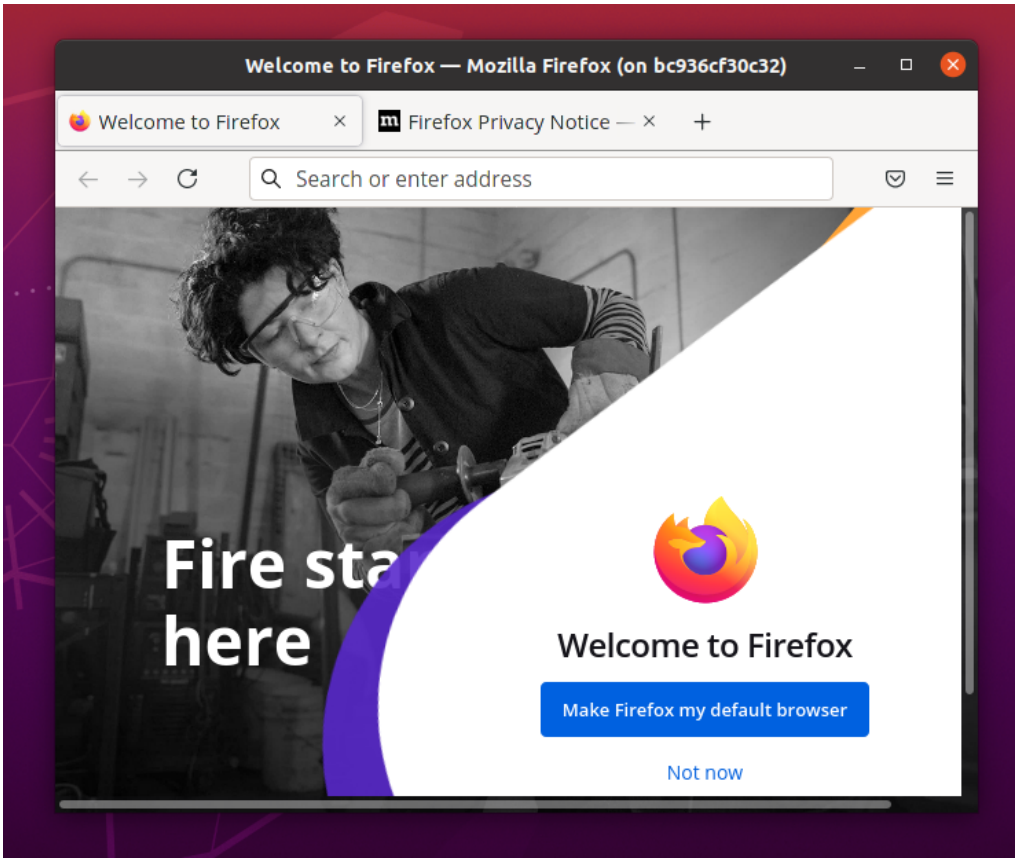

Figure 4.3. Firefox over X11

```
docker cp <container_Id>:/etc/ssh .
```

It is a Docker command that copies the "ssh" folder of the container to the current directory on the host, which is the one containing the Dockerfile.

The updated Dockerfile has the following content:

```
FROM alpine

EXPOSE 22

RUN apk update \
  && apk add --no-cache openssh openrc firefox xauth libcanberra-gtk3 \
  ttf-opensans \
  && mkdir -p /run/openrc \
  && touch /run/openrc/softlevel

RUN adduser user -D
RUN echo "user:password" | chpasswd

ENTRYPOINT ["sh","-c","rc-status; rc-service sshd start; sh"]
```

The `EXPOSE` command is just declarative and informs Docker that the container exposes a service on port 22, which is SSH. The packages installed are the same as before. Then OpenRC is set up. A new user called "user" is added to the container with `useradd` and right after the password for that account is set to "password" with the command `chpasswd`. Setting the password statically in the Dockerfile is not a good practice but it's easier for demonstration purposes. Also it can be changed inside the container at run-time. Finally the `ENTRYPOINT` command enables

the sshd service and spawns a shell on the container when it starts. This shell runs as root but is only accessible from the Docker host for configuration purposes if needed.

Then, after rebuilding the new image with the same build command, to start the container and mount the "ssh" folder inside it, the command is

```
sudo docker run --name firefox -it -v $(pwd)/ssh:/etc/ssh:ro firefoxalpine:1.0
```

The option `-v $(pwd)/ssh:/etc/ssh:ro` mount the "ssh" folder present in the current directory (`pwd`) inside the container at the path `/etc/ssh` as read-only (`:ro`). This folder contains the host keys of the container and the already prepared configuration files.

4.4 IDS with Snort

Snort is an open source IDS and IPS which uses a customizable set of rules to detect malicious network activity in the traffic of packets generating alerts and logs. Snort has three main modes:

- Sniffer mode: reads packets transiting on the network and prints them on the console in a continuous stream;
- Packet logger: logs the packets;
- IDS/IPS mode: performs analysis and detection of the network traffic in a configurable way.

4.4.1 Snort rules

Snort works with rules which are defined in configuration files. The IDS analyzes the traffic and if a packet matches a rule then a customizable action is performed. Rules can be downloaded from the Snort website or can be custom made and have a particular syntax that needs to be followed. This syntax is divided into:

- Action: alert, log, pass, drop, reject, sdrop;
- Protocol: tcp, udp, icmp;
- Source address and port;
- Direction: "->" or "<-";
- Destination address and port;
- Rule options: msg, sid, logto and others.

The action is the countermeasure to realize when a packet matches the rule, for example "alert" generates an alert and logs the packet and "drop" blocks and logs the packet. Rule options are a set of further specifications to define a rule, for example "msg" allows to specify the message to print on the screen or "sid" defines the snort rule identification number. Here's an example of a rule:

```
alert icmp any any -> 8.8.8.8 any (msg:"Pinging...";sid:1000004;)
```

This rule prints an alert on the console with the message "Pinging..." whenever an icmp packet coming from any address and port and directed to the IP 8.8.8.8 on any port is detected.

4.4.2 Snort in a container

To run Snort inside a container it's necessary to have an image. There are third parties custom images already built on the DockerHub but Snort does not have an official repository with official images. This section shows how to build the image starting from a base OS layer.

There are different methods to install Snort, Alpine offers a simple package installation which requires some configuration files to be added. Moreover since Alpine is the lightest base layer operating system it is a good choice.

All the configurations and rules files can be downloaded from the Snort site at https://www.snort.org/downloads/registered/snortrules-snapshot-2983.tar.gz where the last numbers are the version of Snort. The `snort.conf` file is the core of the Snort configuration and requires some other configuration and rules files to work. It is structured in sections and contains a list of environment variables that should be set according to the situation, like `PATH` that needs to specify the paths to the various folders. The file also includes the list of rules files that will be used from the IDS. The rules that are not used should be commented out. A more careful analysis of this file should be done by users to get a better overview of the tool. The Dockerfile created in this section includes just a minimal set of files for demonstration purposes.

In a new directory two folders "config" and "rules" are created. The first one should include the files `classification.config, reference.config, threshold.conf, unicode.map, snort.conf` which can be found in the archive already downloaded from the site. The second folder should contain the file `local.rules`. This file contains a single rule to illustrate how Snort works:

```
alert icmp any any -> any any (msg:"Pinging...";sid:1000004;)
```

This rule prints an alert on the console every time an icmp packet is intercepted. The resulting Dockerfile is:

```
FROM alpine

RUN apk add update && apk add --no-cache snort

COPY config/classification.config /etc/snort/classification.config
COPY config/reference.config /etc/snort/reference.config
COPY config/threshold.conf /etc/snort/threshold.conf
COPY config/unicode.map /etc/snort/unicode.map
COPY config/snort.conf /etc/snort/snort.conf

RUN mkdir /etc/snort/rules
COPY rules/local.rules /etc/snort/rules/local.rules

CMD ["/bin/sh"]
```

The path to the rules files is defined in the `snort.conf` file and can be modified as desired. The `CMD` spawns a shell in the container. At this point the command to build the image from the Dockerfile directory is:

```
sudo docker build . -t snort:1.0
```

Then to run the image in interactive mode and attach the container shell:

```
sudo docker run -it snort:1.0
```

The container is connected to the default bridge Docker0. To test Snort with the rule in `local.rules` the command to run inside the container is:

```
snort -A console -q -c /etc/snort/snort.conf -i eth0
```

This command runs snort in IDS mode and prints alerts on the console (`-A console`) without showing the banner and status report (`-q` for quiet mode) using the `snort.conf` file on the interface `eth0`. Now to test Snort the container needs to receive icmp packets, for example a simple `ping` from the host to the container IP should work:

```
user@user:~$ ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.317 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.215 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.185 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.185 ms
64 bytes from 172.17.0.2: icmp_seq=5 ttl=64 time=0.185 ms
```

If the process was successful the alert messages should appear on the Snort console inside the container as shown below.

```
/ # snort -A console -q -c /etc/snort/snort.conf -i eth0
11/22-17:19:13.665009 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.1 -> 172.17.0.2
11/22-17:19:13.665054 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.2 -> 172.17.0.1
11/22-17:19:14.675384 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.1 -> 172.17.0.2
11/22-17:19:14.675452 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.2 -> 172.17.0.1
11/22-17:19:15.699354 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.1 -> 172.17.0.2
11/22-17:19:15.699415 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.2 -> 172.17.0.1
11/22-17:19:16.723462 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.1 -> 172.17.0.2
11/22-17:19:16.723523 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.2 -> 172.17.0.1
11/22-17:19:17.747415 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.1 -> 172.17.0.2
11/22-17:19:17.747475 [**] [1:1000004:0] Pinging... [**] [Priority: 0] {ICMP}
    172.17.0.2 -> 172.17.0.1
```

The number of alerts is double the number of pings because Snort prints out also the ping reply. This is a simple demonstration on the use of Snort inside a container. To set up an efficient IDS a lot of rules could be implemented to cover various aspects of network security. Snort offers three sets of rules ready to use: “Community Ruleset”, ”Registered Ruleset” and ”Snort Subscriber Ruleset”. The first two sets are free of charge and the last one requires a paid subscription. All of them can be downloaded from the Snort website. The difference between the Registered and the Subscriber rulesets is a time difference of thirty days in the release of official rules, while the Community ruleset is submitted by members of the Snort community. The rules chosen for a project must be included in the image by changing the Dockerfile and rebuilding the image adding the new rules exactly as the `local.rules` file was inserted in the previous Dockerfile. Also before building the image the file `snort.conf` should be modified according to the rules files added, by inserting or uncommenting the line relative to those files under the section `7)Customize your rule set`.

For testing purposes the Snort container was directly pinged but in the final project all the traffic in transit on the network will be analyzed by mirroring it on the Snort container interface.

4.4.3 Snort logs

When Snort is active in IDS mode its operations are continuously logged. The default path to the log files is `/var/log/snort/`. There are two types of log file: the files that contain the information on the individual packets intercepted and the files that contain other general information. The first type are PCAP files, which is a specific format to record packet data from a network scan, and can be read with the command `snort -r <file-name>`. The others are standard text files.

4.5 Switch and internal Firewall

The network topology defined for this project includes a switch and a firewall. The switch is a layer 2 network device that is used to connect hosts and containers on the same network, making them reachable to each other. In the project addressed in this document it is used to connect all the containers on the same network. The firewall is a network security software or hardware component that monitors and manages incoming and outgoing traffic based on internal security rules by creating a barrier between two networks, typically the Internet (WAN) and an internal private network (LAN).

While the default firewall offered by OpenWrt acts as a perimeter firewall creating a barrier between the internal LAN and the WAN (Internet), the firewall created in this section monitors and manages the traffic on the internal LAN, adding an extra layer of security. The switch shown in this section allows the connection of all containers within the LAN to each other and to the router. Switch and internal firewall are placed in the same container to allow the passage and monitoring of internal traffic. The firewall should be set to allow only internally expected traffic to pass through, based on the services implemented in the application and the operations that users are authorized to perform within the LAN.

The Linux operating system has a lot of virtual networking options that are also used to host containers, it offers different features to realize both the switch and the firewall without the need to download other third parties software: specifically Linux bridges and iptables.

4.5.1 Linux bridges and iptables

A Linux bridge works as a network switch, it forwards packets through interfaces that are connected to it. It is really easy to configure:

```
# ip link add <br-name> type bridge
# ip link set <interface-name> master <br-name>
```

These commands create a bridge named `<br-name>` and then connect the interface `<interface-name>` to it.

Iptables is a command line utility to configure the kernel-level firewall of Linux, based on netfilter modules. For more detailed information see 3.2.2.

4.5.2 Switch and Firewall in a container

In the final project this container will work as a switch between all the other containers in the network, hence to demonstrate the functioning of this single container it's necessary to have the router container already up and running as described in the 4.2.1 router section. The switch container is then started and connected to the router LAN.

As in the other containers the base image will be Alpine Linux, a minimal OS which supports the Linux features, including bridges and Iptables.

The iptables module is not present in the standard Alpine image from Docker Hub but can be easily added in a Dockerfile. Also the iptables firewall rules need to be set before the container starts. For this reason another file called "enablefirewall.sh" has to be present in the same folder of the Dockerfile. The content of the dockerfile is the following.

```
FROM alpine

RUN apk update && apk add --no-cache iptables

COPY enablefirewall.sh /

ENTRYPOINT ["sh", "-c", "/enablefirewall.sh; sh"]
```

This file simply adds the iptables package and copies the "enablefirewall.sh" file inside the Alpine image. Then when the container is started the command `ENTRYPOINT` executes the "enablefirewall.sh" script and spawns a shell inside the container. The firewall file contains all the rules that need to be enabled for the firewall, that vary from case to case. To build the image from the Dockerfile directory:

```
sudo docker build . -t switchfirewall:1.0
```

The resulting image is called "switchfirewall" and tagged "1.0".

The first part of this section explains the switch aspect of the container through the use of a Linux bridge. The second part concentrates on the firewall (iptables and firewall rules file).

Switch explanation

Since the bridge is isolated inside a container and it's not a standard Docker bridge, the Docker networking commands would not work and all the networking configuration is performed manually using a Linux bridge and veth devices. These are also the Linux features that Docker uses underneath the networking commands. The manual configuration exposes clearly all the processes to connect the various containers to the network and also gives more flexibility to build a network suited to this project.

By default a Docker container is limited on the operations it can perform on the network because all the network functions are provided by the shared host kernel. To enable the possibility of manipulating network devices as well as iptables the Linux capability `CAP_NET_ADMIN` needs to be added to the container on start. This capability allows various network related operations. To start the container:

```
sudo docker run --cap-add NET_ADMIN --name firewall --net=none -it
    switchfirewall:1.0
```

This command runs a container named "firewall" from the "switchfirewall:1.0" image with the added `NET_ADMIN` capability and connects the terminal to an interactive shell inside it. The container is started with no network access because it will be connected to the running router container.

From within the container to create a bridge named "br0" and enable it:

```
ip link add br0 type bridge
ip link set br0 up
```

At this point to illustrate the functioning of the bridge, the following actions have to be performed:

1. Start the router container and move the Ethernet interface connected to the WAN inside its namespaces. The procedure has been explained in section 4.2.1.

2. Connect the "br-lan" router bridge to the "br0" bridge inside the "firewall" container with a veth device allowing the switch connection to the LAN;

3. Start another alpine container with no network and connect it to the "br0" bridge with another veth device to test the connection to the router through the switch;

4. Add an IP address on the LAN subnet to the veth endpoint inside the new alpine container and also add the default route through the router gateway;

5. Test the connection between the new alpine container and the router container as well as the Internet connection through the router.

This manual network connection of the three containers with two veth devices has to be done from the host. Most of the following commands and network theory have been explained in section 3.2. Docker creates isolated namespaces for the different containers with different network stacks. To connect two containers a veth needs to be created and the two endpoints need to be moved into the two namespaces and added to the bridge where required.

Before performing this process there is a little modification that needs to be done on the host. When Docker creates a container, it creates a new network namespace located at the path `/proc/<container-id>/ns/net`. Most of the code that is going to be used is based on the `netns` namespaces utility command. This command requires that the namespace files are in the default location `/var/run/netns/<container-name>`. This allows commands like `ip link set <veth-name> netns <container-name>` to work on the correct namespace. To solve this problem it's necessary to create a symlink of the containers namespaces from the first directory to the second one. The commands are:

```
sudo mkdir -p /var/run/netns/
c1pid=$(sudo docker inspect -f '{{.State.Pid}}' <container-name>)
sudo ln -sfT /proc/${c1pid}/ns/net /var/run/netns/<container-name>
```

The first command creates the directory in case it doesn't already exist. The second command extracts the container id in a variable that is used in the third command to create the symlink. This procedure should be done for all the containers whose namespace needs to be manipulated with the `netns` command. At this point the following commands should be clear.

Bringing all together in a bash script:

```
#!/bin/bash

#start router container
docker run --cap-add=NET_ADMIN -dit --name openwrt_1 --network none router:1.0

#start firewall/switch container
docker run --cap-add NET_ADMIN --name firewall --net=none -itd alpine sh

#start alpine test container
docker run --name cont1 --net=none -itd alpine

#create bridge named br inside firewall container
docker exec firewall ip link add name br type bridge
docker exec firewall ip link set br up

#create veth device to connect router and firewall
ip link add veth2a type veth peer name veth2b
ip link set up veth2a
ip link set up veth2b

#symlink firewall container namespace and move veth endpoint inside
fwpid=$(docker inspect -f '{{.State.Pid}}' firewall)
mkdir -p /var/run/netns/
ln -sfT /proc/${fwpid}/ns/net /var/run/netns/firewall
ip link set veth2a netns firewall
ip netns exec firewall ip link set up veth2a

#add moved endpoint to firewall bridge br
docker exec firewall ip link set veth2a master br

#symlink router container namespace and move veth endpoint inside
owpid=$(docker inspect -f '{{.State.Pid}}' openwrt_1)
ln -sfT /proc/${owpid}/ns/net /var/run/netns/openwrt_1
ip link set veth2b netns openwrt_1
```

```
ip netns exec openwrt_1 ip link set up veth2b

#move WAN Ethernet interface to router namespace
ip link set enp6s0 netns openwrt_1
ip netns exec openwrt_1 ip link set up enp6s0

#create veth device to connect test container and firewall
ip link add veth1a type veth peer name veth1b
ip link set up veth1a
ip link set up veth1b

#symlink test container namespace and move veth endpoint inside
c1pid=$(docker inspect -f '{{.State.Pid}}' cont1)
ln -sfT /proc/${c1pid}/ns/net /var/run/netns/cont1
ip link set veth1b netns cont1
ip netns exec cont1 ip link set up veth1b

#add static ip to endpoint interface and default gateway inside test container
ip netns exec cont1 ip addr add 192.168.16.50/16 dev veth1b
ip netns exec cont1 ip route add default via 192.168.16.1

#move veth endpoint inside firewall container and add it to bridge
ip link set veth1a netns firewall
ip netns exec firewall ip link set up veth1a
docker exec firewall ip link set veth1a master br
```

At this point, if everything worked correctly, the three containers should be running in detached mode due to the option `-d` specified in the starting commands, meaning that they are not directly connected to a terminal but running in background. From a host terminal to attach the test container "cont1":

```
sudo docker attach cont1
```

This command attaches the container shell to the current terminal. Now before testing the Internet connection it's convenient to change the DNS configuration. Since the system does not provide a DNS service the container is not able to resolve domain names. A good solution is to use Google's DNS servers. The IP addresses of those servers are 8.8.8.8 and 8.8.4.4. To take advantage of this service the following lines should be added to the file `/etc/resolv.conf` inside the container:

```
nameserver 8.8.8.8
nameserver 8.8.4.4
```

At this point it's possible to test the Internet connection through the router by executing a ping to any Internet server, for example to the URL "www.google.com".

```
/ # ping www.google.com
PING www.google.com (142.250.180.68): 56 data bytes
64 bytes from 142.250.180.68: seq=0 ttl=116 time=24.257 ms
64 bytes from 142.250.180.68: seq=1 ttl=116 time=23.432 ms
64 bytes from 142.250.180.68: seq=2 ttl=116 time=23.268 ms
64 bytes from 142.250.180.68: seq=3 ttl=116 time=23.611 ms
64 bytes from 142.250.180.68: seq=4 ttl=116 time=23.380 ms
^C
--- www.google.com ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max = 22.435/23.389/24.257 ms
```

The connection is active. To show the route of the packets in transit across the network it is possible to use the command `traceroute`:

```
/ # traceroute www.google.com
traceroute to www.google.com (216.58.208.132), 30 hops max, 46 byte packets
 1  192.168.16.1 (192.168.16.1)  0.029 ms  0.027 ms  0.016 ms
 2  192.168.1.1 (192.168.1.1)  1.337 ms  1.720 ms  1.775 ms
 3  10.1.3.131 (10.1.3.131)  167.359 ms  45.792 ms  35.165 ms
```

The first entry shows that the packets are actually transiting through the containerized router whose gateway is at 192.168.16.1. All the components work correctly and the connection is established.

Firewall explanation

The firewall is managed through the script "enablefirewall.sh". This file has to be filled with all the iptables rules to create the desired firewall functionalities. After writing or changing the rules in this script the image needs to be rebuilt in order to update the file inside the new image. When the container is started the script is executed inside it and the rules are applied to the container. Just for demonstration here is reported an example that blocks all the network communications and DROP all the packets. The file contains the following lines.

```
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT DROP
```

Those commands set the three chains INPUT, OUTPUT and FORWARD to drop all the packets incoming, outgoing or in transit on the container. Rebuilding the image and running the container results in a firewall container that blocks all the traffic, so that even a ping command would not work.

The firewall rules should be chosen based on the requirements and the services offered in the network. The rules of the project addressed in this document are illustrated in the next chapter.

4.6 Access Point

The wireless access point allows devices to connect wireless to the LAN. This section demonstrates how to activate an isolated access point inside a Docker container. The process described is based on another project [5] which has been modified according to the application requirements.

4.6.1 Access Point inside a container

To build the container image a Dockerfile is required. This file is created inside a new folder together with another folder called "confs" which contains configuration files. Inside "confs" another folder called "hostapd_confs" is added for other configuration files. The Dockerfile has the following content:

```
FROM alpine

# Install packages
RUN apk update && apk add hostapd iw dhcp vim iptables

# Configure Hostapd
ADD confs/hostapd_confs/wpa2.conf /etc/hostapd/hostapd.conf
# Configure DHCPD
ADD confs/dhcpd.conf /etc/dhcp/dhcpd.conf
RUN touch /var/lib/dhcp/dhcpd.leases

# Configure networking
```

```
ADD confs/iptables.sh /iptables.sh
ADD confs/iptables_off.sh /iptables_off.sh

# Copy init file
ADD confs/start.sh /start.sh
CMD ["/bin/sh"]
```

Starting from the Alpine base image some packages are downloaded, specifically:

- hostapd: the software for the access point;
- iw: software for debugging the wireless interfaces;
- dhcp: dhcp service to give address to clients;
- vim: tool to edit files;
- iptables: Linux software for network routing and firewalling.

Then some files are added to the image. Those files will be discussed later. Finally the `CMD` command spawns a shell when the container is booted.

To configure the access point a file named "wpa2.conf" is needed inside the folder "hostapd_confs". The file has the following content.

```
interface=wlp7s0
driver=nl80211
ssid=apdocker_wpa2
hw_mode=g
ieee80211n=1
channel=6
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
country_code=ES
macaddr_acl=0

wpa_passphrase=password
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP
rsn_pairwise=CCMP
```

This file configures a WPA2 secure access point. The "wpa_passphrase" field contains the network password and the "ssid" field contains the name of the wireless network. The "interface" field contains the name of the wireless card interface which may change from host to host.

The access point provides a DHCP service configured with a file named "dhcpd.conf" in the "confs" folder. The file has the following content.

```
authoritative;
subnet 11.0.0.0 netmask 255.255.255.0 {
       range 11.0.0.10 11.0.0.20;
       option broadcast-address 11.0.0.255;
       option routers 11.0.0.1;
       default-lease-time 600;
       max-lease-time 7200;
       option domain-name "local";
       option domain-name-servers 8.8.8.8;
}
```

This code creates a DHCP server and defines a network on the subnet 11.0.0.0/24 that grants a range of 10 client addresses which can be modified according to the application requirements.

Then there are two iptables configuration scripts in the "confs" folder: "iptables.sh" and "iptables_off.sh". These files contain the iptables rules that implement the NAT to allow the transparent communication between the access point's network and the external network. In this case, the NAT will be created between wlp7s0 and veth4b interfaces, the second serving Internet connection to the first. These interfaces are the two that will be present inside the container: wlp7s0 is the interface of the wireless card to enable the access point and veth4b is the virtual Ethernet interface connected to the switch that will provide the Internet connection.

The "iptables.sh" script contains the rules required to establish the NAT and has the following content:

```
#/bin/sh
iptables-nft -t nat -C POSTROUTING -o veth4b -j MASQUERADE || iptables-nft -t
    nat -A POSTROUTING -o veth4b -j MASQUERADE
iptables-nft -C FORWARD -i veth4b -o wlp7s0 -m state --state
    RELATED,ESTABLISHED -j ACCEPT || iptables-nft -A FORWARD -i veth4b -o
    wlp7s0 -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables-nft -C FORWARD -i wlp7s0 -o veth4b -j ACCEPT || iptables-nft -A
    FORWARD -i wlp7s0 -o veth4b -j ACCEPT
```

On each line it's used the logical operator OR. The result is that for each line the first command (the one on the left of the operator) will be executed. If the command exits successfully (return code 0) the second command (the one on the right of the operator) won't be executed. In case it returns an error (return code 1), the second command will be executed.

On each line, the first command is iptables-nft, which is used on Alpine to manage the Linux firewall system based on nftables (equivalent to iptables), but still using the classic iptables rule syntax. It looks for a rule (`-C`) on the current applied ruleset. If the rule does not exist, the check command returns a value 1 (error), and hence the second command is executed, which actually adds (`-A`) the rule. This prevents rules from being applied twice. As for the rules themselves:

- The first creates a rule on POSTROUTING chain and NAT table over veth4b interface, creating a Masquerade type NAT (this changes the original IP for the one defined on veth4b on the outbound network packets);
- The second rule allows the redirection of the network packets from veth4b interface towards wlp7s0 interface as long as they are related to an already established connection (RELATED,ESTABLISHED);
- The third rule allows the redirection of the network packets from the wlp7s0 interface towards the veth4b interface in any case, allowing the opening of new outbound connections.

The "iptables_off.sh" script is similar and has the following content:

```
#/bin/sh
iptables-nft -t nat -C POSTROUTING -o veth4b -j MASQUERADE && iptables-nft -t
    nat -D POSTROUTING -o veth4b -j MASQUERADE
iptables-nft -C FORWARD -i veth4b -o wlp7s0 -m state --state
    RELATED,ESTABLISHED -j ACCEPT && iptables-nft -D FORWARD -i veth4b -o
    wlp7s0 -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables-nft -C FORWARD -i wlp7s0 -o veth4b -j ACCEPT && iptables-nft -D
    FORWARD -i wlp7s0 -o veth4b -j ACCEPT
```

Note that the content is almost the same except for two details. The first is the substitution of the OR operator for the AND operator. With this, the second command on each line is executed only if the first one does successfully exit, meaning that the rule does exist and hence must be dropped by the script. The second detail is that instead of adding the rules they are deleted using the flag `-D`.

Lastly there is an initial script "start.sh" in the "confs" folder. This script enables all the services and starts the access point. It contains the following lines:

```
#!/bin/sh

NOCOLOR='\033[0m'
RED='\033[0;31m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'

sigterm_handler () {
  echo -e "${CYAN}[*] Caught SIGTERM/SIGINT!${NOCOLOR}"
  pkill hostapd
  cleanup
  exit 0
}
cleanup () {
  echo -e "${CYAN}[*] Deleting iptables rules...${NOCOLOR}"
  sh /iptables_off.sh || echo -e "${RED}[-] Error deleting iptables
      rules${NOCOLOR}"
  echo -e "${GREEN}[+] Successfully exited, byebye! ${NOCOLOR}"
}

trap 'sigterm_handler' TERM INT
echo -e "${CYAN}[*] Creating iptables rules${NOCOLOR}"
sh /iptables.sh || echo -e "${RED}[-] Error creating iptables rules${NOCOLOR}"

echo -e "${CYAN}[+] Configuration successful! Services will start
    now${NOCOLOR}"
dhcpd -4 -f -d wlp7s0 &
hostapd /etc/hostapd/hostapd.conf &
pid=$!
wait $pid

cleanup
```

The first lines declare some ANSI color code in order to have a more colorful output and make it more visually intuitive. Then the "sigterm_handler" function is defined, which is used to catch SIGTERM and SIGINT signals that the process might receive by sending a keyboard interruption on interactive mode (CTRL+C) or by stopping the container via Docker commands. This handler is enabled with the line `trap 'sigterm_handler' TERM INT`. The function will stop the execution of the hostapd service and will execute the "cleanup" function. Then the "cleanup" function is declared to bring back the container into the initial state removing the iptables rules with the script "iptables_off.sh". After declaring the signal trap with "sigterm_handler" the iptables.sh script is executed in order to create the network iptables rules. Lastly, the services themselves are executed:

- The dhcpd service is executed with parameters -4 -f -d wlp7s0, indicating that IPv4 addresses are used, it is executed on the foreground (the logs are shown) and the logs are sent to stderr, being executed explicitly on wlp7s0. Also there is the `&` character in the end, making the process run in the background. This might seem to contradict -f parameter, but the result is that even if the process runs in the background the logs are printed on the screen;
- The hostapd service is executed, also in the background making use of `&`, reading the configuration stored on /etc/hostapd/hostapd.conf which in this case enables a WPA2 protected access point.

In the end, the script stores the PID of the hostapd process and waits until it is finished. If

hostapd fails at any moment and terminates the execution, the cleanup function will always allow the container to return to its initial state.

All the configuration files are created and everything is set to build the Docker image. To build the image from the Dockerfile directory the command is:

```
sudo docker build . -t accesspoint:1.1
```

To illustrate the functioning of the container it needs to be connected to the LAN through the switch container illustrated in the previous section. This means that before executing the following commands the custom network composed by router and switch containers needs to be already up and running. The process to activate the custom network and to connect containers to the LAN is explained in section 4.5.2.

The following commands execute the container and connect it to the LAN network:

```
#start access point container
sudo docker run --name ap --cap-add=NET_ADMIN -dit --network=none
    accesspoint:1.1

#create veth device to connect accesspoint and switch/firewall containers
sudo ip link add veth4a type veth peer name veth4b
sudo ip link set up veth4a
sudo ip link set up veth4b

#move veth end point inside firewall/switch container and add it to bridge
    (symlink already done)
sudo ip link set veth4a netns firewall
sudo ip netns exec firewall ip link set up veth4a
sudo docker exec firewall ip link set veth4a master br

#symlink accesspoint container namespace and move veth endpoint inside
PID=$(sudo docker inspect -f '{{.State.Pid}}' ap)
sudo ln -sfT /proc/${PID}/ns/net /var/run/netns/ap
sudo ip link set veth4b netns ap
sudo ip netns exec ap ip link set up veth4b

#add static ip to endpoint interface and default gateway inside access point
    container
sudo ip netns exec ap ip addr add 192.168.16.70/16 dev veth4b
sudo ip netns exec ap ip route add default via 192.168.16.1
```

The first command executes an access point container named "ap" with the added Linux capability NET_ADMIN allowing the container to perform network related commands. The options `-dit` means that the container is interactive and detached from the current terminal. The other commands connect the container to the LAN (see 4.5.2). Note that the name of the veth end point moved into the access point container should match the name of the interface used during the image configuration which in this case is "veth4b".

By default the wireless card interface wlp7s0 is in the default host namespace. Before accessing the container and activating the access point the interface needs to be moved from the host namespace to the container namespace with the following commands:

```
PID=$(sudo docker inspect -f '{{.State.Pid}}' ap)
PHY=$(sudo cat /sys/class/net/wlp7s0/phy80211/name)
sudo iw phy $PHY set netns $PID
```

The first command gets the container ID. The second one gets the physical device associated with the wireless interface and the third moves the interface to the container namespace. At this point the interface is set but has no IP address which needs to be statically configured. The IP needs to be set to 11.0.0.1/24 to match the dhcpd.conf file. Also it is good practice to bring up the interface in case it is down.

```
sudo docker exec ap ip link set up wlp7s0
sudo docker exec ap ip addr add 11.0.0.1/24 dev wlp7s0
```

At this point it is possible to test the access point by attaching the container shell to the current terminal:

```
sudo docker attach ap
```

Finally to enable the access point the "start.sh" script must be executed.

```
./start.sh
```

If the process was successful the following lines should appear on the screen.

```
/ # ./start.sh
[*] Creating iptables rules
iptables: Bad rule (does a matching rule exist in that chain?).
iptables: Bad rule (does a matching rule exist in that chain?).
iptables: Bad rule (does a matching rule exist in that chain?).
[*] Setting wlp7s0 settings
[+] Configuration successful! Services will start now
Configuration file: /etc/hostapd/hostapd.conf
Internet Systems Consortium DHCP Server 4.4.2-P1
Copyright 2004-2021 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/
Config file: /etc/dhcp/dhcpd.conf
Database file: /var/lib/dhcp/dhcpd.leases
PID file: /run/dhcp/dhcpd.pid
Wrote 1 leases to leases file.
rfkill: Cannot open RFKILL control device
wlp7s0: interface state UNINITIALIZED->COUNTRY_UPDATE
Using interface wlp7s0 with hwaddr b8:76:3f:4e:43:25 and ssid "apdocker_wpa2"
Listening on LPF/wlp7s0/b8:76:3f:4e:43:25/11.0.0.0/24
Sending on   LPF/wlp7s0/b8:76:3f:4e:43:25/11.0.0.0/24
Sending on   Socket/fallback/fallback-net
Server starting service.
wlp7s0: interface state COUNTRY_UPDATE->ENABLED
wlp7s0: AP-ENABLED
```

The access point is enabled and now other devices can be connected through WiFi. When a device is connected to the network the connection process is shown on the screen. The same when it is disconnected:

```
wlp7s0: STA 1e:3e:bd:70:ef:8b IEEE 802.11: authenticated
HT: Forty MHz Intolerant is set by STA 1e:3e:bd:70:ef:8b in Association
    Request
wlp7s0: STA 1e:3e:bd:70:ef:8b IEEE 802.11: associated (aid 1)
wlp7s0: AP-STA-CONNECTED 1e:3e:bd:70:ef:8b
wlp7s0: STA 1e:3e:bd:70:ef:8b RADIUS: starting accounting session
    EB54B7146E5BE0E0
wlp7s0: STA 1e:3e:bd:70:ef:8b WPA: pairwise key handshake completed (RSN)
DHCPDISCOVER from 1e:3e:bd:70:ef:8b via wlp7s0
DHCPOFFER on 11.0.0.10 to 1e:3e:bd:70:ef:8b via wlp7s0
DHCPREQUEST for 11.0.0.10 (11.0.0.1) from 1e:3e:bd:70:ef:8b via wlp7s0
DHCPACK on 11.0.0.10 to 1e:3e:bd:70:ef:8b via wlp7s0
reuse_lease: lease age 0 (secs) under 25% threshold, reply with unaltered,
    existing lease for 11.0.0.10
DHCPREQUEST for 11.0.0.10 (11.0.0.1) from 1e:3e:bd:70:ef:8b via wlp7s0
```

```
DHCPACK on 11.0.0.10 to 1e:3e:bd:70:ef:8b via wlp7s0
wlp7s0: AP-STA-DISCONNECTED 1e:3e:bd:70:ef:8b
wlp7s0: STA 1e:3e:bd:70:ef:8b IEEE 802.11: disassociated
wlp7s0: STA 1e:3e:bd:70:ef:8b IEEE 802.11: deauthenticated due to inactivity
    (timer DEAUTH/REMOVE)
```

When a device is connected to the access point it joins the LAN network created in the previous sections and, besides being connected to the Internet, it can reach the other devices or containers connected.

Chapter 5

Final project

5.1 Introduction

The application is composed of all the components individually analyzed in the previous chapter. Those containers are interconnected to build a network providing all their services. This section illustrates the application characteristics and integrations.

5.1.1 Topology

The application is composed of the following services running inside containers: a router, a switch, a perimeter firewall and an internal firewall, an access point, an IDS and a container to run a web browser using X11 over SSH. Those services are connected to each other in a precise topology shown in the image 5.1.

5.1.2 Programming language

The programming language chosen for the application is Bash. The reasons that led to the adoption of this language are many. The main one is the direct interaction with the Linux system. All the commands used, the Docker APIs, the Linux network settings and the kernel-level operations are natively connected to the Bash shell and language. Also all the containers implement Linux as a base operating system. The Bash language offers all the commands and options needed for the implementation of the application making it the most suitable choice.

5.1.3 Setting the application firewall rules

In section 4.5.2 the creation and use of the container that provides the switch and the internal firewall has been explained but no firewall rules have actually been inserted to enable the firewall. The reason is that the firewall rules need to be adapted to the situation and can be different based on the application services and requirements. This section illustrates a possible configuration created for the addressed application.

The rules must be inserted in the “enablefirewall.sh” file relative to the switch/firewall image. Every time the file is modified the image needs to be rebuilt to incorporate the new configuration.

In the case of the application realized in this document the internal firewall must allow only the traffic expected by the containers through their interfaces connected to the switch. Only traffic generated by these legitimate containers should be able to reach the router and only connections generated from within the LAN to the external network should be allowed. The rules to be inserted in the firewall file are:

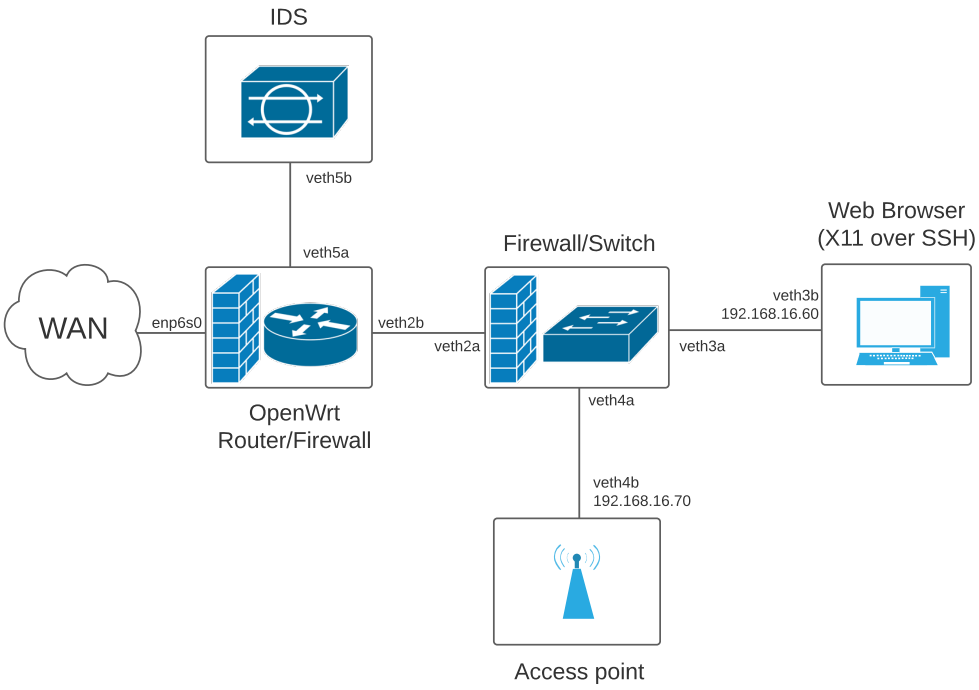

Figure 5.1. Application topology

```
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -A FORWARD -m physdev --physdev-in veth2a --physdev-out veth3a -m
    state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -m physdev --physdev-in veth2a --physdev-out veth4a -m
    state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -m physdev --physdev-in veth2a -j DROP
```

In this container the packets are in transit to reach other destinations through the switch, meaning that the INPUT and OUTPUT iptables chains won't be used. The first two lines DROP all the packets matching these chains. The only chain used is FORWARD as it matches the packets that are transiting the container to reach other hosts. The third and fourth rule allows the Internet connection or more in general any traffic which has been requested from inside the LAN. Specifically `physdev` is an iptables module that manages the devices enslaved to a bridge, `physdev-in` and `physdev-out` identify respectively the interfaces from where packets are coming and going out. The `ESTABLISHED` and `RELATED` flags allow incoming packets associated with a connection which has seen packets in both directions or packets starting a new connection, but associated with an existing one. In simple words packets are accepted only if the connection is started from within the internal network. The connection can be started either from the devices connected to the access point (coming from veth4a) or the web browser on the "firefox" container (coming from veth3a). The last rule blocks any other connection that doesn't match the previous ones.

This set of rules can be expanded according on the services provided by the network.

5.1.4 Configuration variables

In the application two system network interfaces of the underlying system are used. The first is the Ethernet interface of the WAN which provides the Internet connection, the second is the wireless

card interface to enable the access point. Considering the computer used for the development and testing of the application the name of the two interfaces are respectively "enp6s0" and "wlp7s0". Unfortunately those interface names are not equal on each system. Different systems may have different names and if the wrong interfaces are used on another computer then the application would not work.

This problem can be easily solved by adding to the script two variables that contain the names of the two interfaces and then use those variables to configure the entire application. This method allows each different user to use the same images and scripts just by changing those variables to the correct values on its system. The operation is performed as follows. First the two variables are created:

```
INT='enp6s0'
WC='wlp7s0'
```

Those variables store respectively the Ethernet interface and the wireless card interface names. In this case the names are "enp6s0" and "wlp7s0".

At this point there are two cases, the occurrences of the interface names are either inside the application script or inside the containers that will be started when the script is executed. In the first case any occurrence of the name is replaced with the relative variable preceded by `$` to retrieve its value. For example:

```
ip link set enp6s0 netns openwrt_1
```

becomes

```
ip link set $INT netns openwrt_1
```

Changing the occurrences of names inside the containers is a little more difficult. Two functions are used, the first is the Docker API `docker exec CONTAINER COMMAND` which allows to execute commands on a running container, the second is the Linux command `sed` which performs basic text transformation on files. For example to substitute any occurrences of a text (`<text1>`) with another text (`<text2>`) inside a file stored in a running container the command would be:

```
sudo docker exec <container-name> sed -i "s/<text1>/<text2>/g" <path-to-file>
```

In the application the two containers that include the interfaces names that need to be changed are the access point and the router. In particular the router includes the "enp6s0" interface inside the network configuration file and the access point includes the wireless card interface "wlp7s0" in four files. For example, considering the access point container, to modify the interface name instances with the `WC` variable defined the command would be:

```
sudo docker exec ap sed -i "s/wlp7s0/$WC/g" /etc/hostapd/hostapd.conf /iptables.sh
/iptables_off.sh /start.sh
```

As shown it is possible to insert multiple files separated by a space. This command should be inserted in the script after the container starts.

The same procedure of creating a variable to change some configuration values can be used also for other components to make the application customizable. For example an option that, in addition to increasing the application functionality also increases its security, is the possibility to choose the WI-FI password and SSID. The process to implement this option is the same as before, creating a variable:

```
WIFIPASSWORD='password'
```

and then change the old password occurrences inside the access point container after starting it:

```
sudo docker exec ap sed -i "s/password/$WIFIPASSWORD/g" /etc/hostapd/hostapd.conf
```

Same for the SSID.

5.1.5 Host connection

During application execution, the Ethernet (enp6s0) and wireless card (wlp7s0) interfaces are moved from the default network namespace to the containers namespaces. Since these interfaces allow the host to connect to the Internet, the host remains disconnected from any network when the application is running. This behavior brings both advantages and disadvantages. From a security point of view, the host is isolated from any network and therefore cannot be attacked directly. In this way, however, it can no longer perform operations that require Internet access, such as updating images by rebuilding them. In any case, if necessary, the host can be connected to the LAN created by the application exactly as if it were a container by creating a veth, leaving one endpoint in the host namespace and connecting the other one to the switch. Also, the host must be given a new IP and default route to join the network.

5.2 Application

The application has been built and tested on a computer running Ubuntu 20.04 LTS with x86-64 architecture. Two manuals have been written as documentation: Developer and User manual.

The Developer manual illustrates and explains the implementation of the application, the scripts, the material to build the images and the possible configuration choices. To see the manual refer to 8.1.

The User manual focuses on the application usage for a normal user. It illustrates the procedure to start the application, the system requirements and how to use the services provided by the application. To see the manual refer to 7.1

Chapter 6

Experimental evaluation

This chapter is dedicated to the experimental evaluation of the application developed in this document. The analysis is focused on four aspects:

- The speed of the Internet connection inside the custom network compared to the one of a normal container outside the network on the host;
- The total size of the Docker images used by the application;
- The dynamic memory and CPU consumption of the containers when the application is running;
- The network throughput;

Those aspects are even more important if the application is deployed on an embedded system with limited hardware resources. After these evaluations, the last section focuses on the implementation and testing of a port scan detector using the IDS provided by the application to increase network security.

6.1 Internet speed comparison

To evaluate the speed of the Internet connection inside the custom LAN created by the application two containers are taken into consideration. The first is the "firefox" container created inside the LAN. The second is a simple alpine container started on the host that is not connected to the custom LAN. From the application LAN perspective, this last container is directly connected to the WAN which, for testing and developing purposes, is the home router LAN already present in the local environment. Moreover, in addition to those two containers, another measure of the Internet speed is taken from a device connected wireless to the custom access point created in the application which therefore is also connected to the application LAN.

The speed is tested with a tool called "SpeedTest" provided by the Oockla [6] company. This tool can be used either with a web browser or with a command line package. In the alpine containers the tool is used through the command line. The command to download the package is `apk add speedtest-cli` and to run the test `speedtest`. The process takes some seconds and when it terminates the following lines should be shown on the screen:

```
/ # speedtest
Retrieving speedtest.net configuration...
Testing from Fastweb (93.35.147.242)...
Retrieving speedtest.net server list...
Selecting best server based on ping...
Hosted by HAL Service SPA (Milano) [126.14 km]: 118.494 ms
```

```
Testing download speed.....................................................
Download: 17.43 Mbit/s
Testing upload
    speed.....................................................
Upload: 1.13 Mbit/s
```

Alternatively it is possible to run the tool at the site https://www.speedtest.net using a web browser.

The purpose of the test is to compare the Internet speed of a container or device connected to the application LAN with the speed of one which instead is not connected. The test is not meant to check the absolute Internet speed of the application network rather to compare the relative speed inside and outside the network.

The Internet speed comparison is performed on the download speed, measured in Mbit/s.

Since the Internet connection speed is variable throughout the day depending on different factors, performing just one measure for each system could lead to inaccurate results. Instead an average of five measures for each system is performed to obtain a more precise evaluation. The measures are illustrated in table 6.1.

container LAN	*container WAN*	*device LAN*
17.40	17.54	17.56
16.79	17.36	17.81
17.63	17.83	17.57
17.54	17.71	17.90
17.47	17.47	17.79

Table 6.1. Measures of Internet speed.

The resulting averages of the five measures for each system are shown in table 6.2.

container LAN	*container WAN*	*device LAN*
17.37	17.58	17.73

Table 6.2. Measures of Internet speed.

The resulting averages of the measures in the three cases are quite similar. This means that the network created by the application does not decrease the performance in terms of Internet speed.

6.2 Images total size

The images used to run the containers inside the application need to be stored on the host when the application is started otherwise the containers could not be executed. Every image occupies a certain space on the disk. This could be a problem on systems with limited resources. This section illustrates the total size of the images used and therefore the disk space used by them. The images considered are those built as explained in this document. Any changes performed to the images may lead to a variation in their size.

The command that shows the list of the images on the host as well as their size is `sudo docker images`. The output is the following:

```
router                  thesis      1d5be637c3a5    2 months ago    8.29MB
switchfirewall          thesis      ad52ba6bffa3    13 days ago     9.96MB
firefoxalpine           thesis      f347906f164a    2 months ago    313MB
accesspoint             thesis      c8f8d7a2c39b    2 months ago    34MB
ids                     thesis      6bf990d69a57    7 weeks ago     44.5MB
```

The resulting disk space used by those images is 409.75 MB.

Other files including the application scripts and the material to build the images have not been considered in the total size calculation because their size is so small that it can be overlooked (less than 0.5 MB)

6.3 Resources evaluation at runtime

When the application is executed the containers are started. Each container is an active process that uses dynamic resources such as RAM and CPU. The consumption of these resources varies depending on what is being executed inside the container. To display a live stream of containers resources usage statistics Docker offers the command `sudo docker stats`. The following informations are shown as live stream columns in the output for each container running:

CONTAINER ID and NAME are the ID and name of the container;

CPU % and MEM % are the percentage of the host's CPU and memory the container is using;

MEM USAGE / LIMIT are the total memory the container is using, and the total amount of memory it is allowed to use;

NET I/O is the amount of data the container has sent and received over its network interface;

BLOCK I/O is the amount of data the container has read to and written from block devices on the host;

PIDS is the number of processes or threads the container has created.

This section analyzes the RAM and CPU consumption of the application addressed in this document. It's not possible to give an absolute evaluation since the resources consumption of the application depends on different factors such as how many devices are connected to the local network and the operations every device is performing on the network. For this reason three different situations are analyzed to give an idea of the application resources consumption. The cases analyzed are the following.

- The application is running and no device is connected to the LAN nor any operation is performed;
- The application is running and one device is connected to the LAN. This device is connected to the "firefox" container over SSH and is using the web browser GUI forwarded with X11;
- The application is running, one device is connected and the IDS is enabled.

It's important to consider that the output of the command `sudo docker stats` is a stream, meaning that it continuously changes based on resource consumption variations. The outputs here reported are just a screenshot of one instant of the stream to give an idea of how many resources the containers are using. The NET I/O and BLOCK I/O columns are omitted because they are not analyzed in this section.

Also note that the limitations on resource usage performed on the containers through the docker commands have been removed to present true consumption values.

First case

The application is started and left idle without performing any operation. The resources statistics are:

```
CONTAINER ID   NAME        CPU %     MEM USAGE / LIMIT     MEM %     PIDS
00f016088e48   ids         0.00%     380KiB / 3.435GiB     0.01%     1
70ad7315430f   ap          0.50%     5.461MiB / 3.435GiB   0.16%     4
bc7271c492e6   firefox     0.00%     1.57MiB / 3.435GiB    0.04%     2
183f09b7afa8   firewall    0.00%     616KiB / 3.435GiB     0.02%     1
35813a5f9eb3   openwrt_1   0.01%     6.16MiB / 3.435GiB    0.18%     9
```

As shown the resources usage in this case is very limited. The "ap" and "openwrt_1" containers consume a very limited quantity of CPU which slightly varies around the values reported. The other containers do not use any CPU and the overall memory consumption is rather limited.

Second case

The application is started and one device is connected wireless to the access point. The device is connected with SSH to the "firefox" container and is using the web browser inside it. The web browser GUI is forwarded using X11 over SSH to the device. The application is therefore performing some services and the resources statistics are:

```
CONTAINER ID   NAME        CPU %     MEM USAGE / LIMIT     MEM %     PIDS
00f016088e48   ids         0.00%     380KiB / 3.435GiB     0.01%     1
70ad7315430f   ap          0.31%     5.504MiB / 3.435GiB   0.16%     4
bc7271c492e6   firefox     139.17%   247.1MiB / 3.435GiB   7.03%     136
183f09b7afa8   firewall    0.00%     616KiB / 3.435GiB     0.02%     1
35813a5f9eb3   openwrt_1   0.01%     6.168MiB / 3.435GiB   0.18%     9
```

The consumption is quite similar to the previous one except for the "firefox" container which is using a lot of CPU and a 7% of RAM. This is due to the Firefox web browser running inside the container which consumes lots of the available resources. The percentage of the CPU is higher than 100% because Docker considers a value of 100% for each CPU core. Docker containers by default are not limited on the resources they can use but in the application some options are added to limit them when the containers start.

Third case

The application is started, one device is connected to the access point and the Snort IDS is enabled. The command used to start the IDS in the container is `snort -A console -q -c /etc/snort/snort.conf -i veth5b`. This command enables Snort in IDS mode using all the rules file specified in the file "snort.conf".

```
CONTAINER ID   NAME        CPU %     MEM USAGE / LIMIT     MEM %     PIDS
00f016088e48   ids         0.07%     242.8MiB / 3.435GiB   6.90%     3
70ad7315430f   ap          0.63%     5.547MiB / 3.435GiB   0.16%     4
bc7271c492e6   firefox     0.00%     25.19MiB / 3.435GiB   0.72%     17
183f09b7afa8   firewall    0.00%     584KiB / 3.435GiB     0.02%     1
35813a5f9eb3   openwrt_1   0.08%     4.234MiB / 3.435GiB   0.12%     9
```

As expected the measures of the IDS container are changed, the CPU consumption is slightly higher and the RAM usage is 6%. The other containers' statistics remain unchanged. It must be noticed that the IDS consumption depends on the workload of traffic that it has to analyze. Moreover when the IDS is started and when it is stopped the CPU consumption reaches higher peaks for short time intervals.

6.4 Throughput performance testing

The throughput of a network is the amount of data successfully transmitted in a unit of time over a communication channel on the network. It is usually measured in bits per second (bit/s). The software used to calculate this value is iPerf3 [8], which is a cross-platform tool that can produce standardized performance measurements for any network.

The main network component that can slow down throughput is the firewall. Since the network created by the application includes two firewalls, the default OpenWrt firewall and the internal firewall in the switch container, in this section four measurements are performed: the first with both firewalls enabled, the second and third with only one firewall enabled and the other disabled and the last with both firewalls disabled. This allows the measurements to reflect the influence of the firewalls on network throughput.

The data flow is measured between the web browser container and the OpenWrt container. To perform this operation the iPerf3 package must be downloaded to both containers with the command `apk add iperf3` in the web browser container and `opkg install iperf3` in the OpenWrt container. Then the command `iperf3 -s` must be executed in the OpenWrt container to activate the server that will receive the data. Instead, for each of the four cases considered, from the web browser container the throughput calculation is started with the `iperf -c 192.168.16.1` command that will start sending packets to the server. By default, the TCP protocol is used. The process performs a 10 second measurement showing progress every second and finally calculating an average over those measurements.

First case: both firewalls enabled

```
/ # iperf3 -c 192.168.16.1
Connecting to host 192.168.16.1, port 5201
[ 5] local 192.168.16.60 port 58118 connected to 192.168.16.1 port 5201
[ ID] Interval Transfer Bitrate Retr Cwnd
[ 5] 0.00-1.00 sec 422 MBytes 3.54 Gbits/sec 0 1.10 MBytes
[ 5] 1.00-2.00 sec 415 MBytes 3.48 Gbits/sec 0 1.72 MBytes
[ 5] 2.00-3.00 sec 424 MBytes 3.55 Gbits/sec 0 2.30 MBytes
[ 5] 3.00-4.00 sec 418 MBytes 3.51 Gbits/sec 0 2.67 MBytes
[ 5] 4.00-5.00 sec 419 MBytes 3.52 Gbits/sec 0 2.94 MBytes
[ 5] 5.00-6.00 sec 425 MBytes 3.56 Gbits/sec 0 3.08 MBytes
[ 5] 6.00-7.00 sec 420 MBytes 3.52 Gbits/sec 0 3.08 MBytes
[ 5] 7.00-8.00 sec 421 MBytes 3.54 Gbits/sec 0 3.08 MBytes
[ 5] 8.00-9.00 sec 422 MBytes 3.54 Gbits/sec 0 3.08 MBytes
[ 5] 9.00-10.00 sec 431 MBytes 3.62 Gbits/sec 0 3.08 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval Transfer Bitrate Retr
[ 5] 0.00-10.00 sec 4.12 GBytes 3.54 Gbits/sec 0 sender
[ 5] 0.00-10.00 sec 4.12 GBytes 3.54 Gbits/sec receiver
```

The resulting throughput when both the firewalls are enabled is 3.54 Gbit/s.

Second case: only OpenWrt firewall enabled

```
/ # iperf3 -c 192.168.16.1
Connecting to host 192.168.16.1, port 5201
[ 5] local 192.168.16.60 port 58234 connected to 192.168.16.1 port 5201
[ ID] Interval Transfer Bitrate Retr Cwnd
[ 5] 0.00-1.00 sec 426 MBytes 3.57 Gbits/sec 0 607 KBytes
[ 5] 1.00-2.00 sec 433 MBytes 3.63 Gbits/sec 0 1.78 MBytes
[ 5] 2.00-3.01 sec 425 MBytes 3.54 Gbits/sec 0 2.17 MBytes
[ 5] 3.01-4.00 sec 432 MBytes 3.65 Gbits/sec 0 2.51 MBytes
```

```
[ 5] 4.00-5.00 sec 434 MBytes 3.65 Gbits/sec 0 2.91 MBytes
[ 5] 5.00-6.00 sec 436 MBytes 3.66 Gbits/sec 0 3.06 MBytes
[ 5] 6.00-7.00 sec 435 MBytes 3.65 Gbits/sec 0 3.06 MBytes
[ 5] 7.00-8.00 sec 435 MBytes 3.65 Gbits/sec 0 3.06 MBytes
[ 5] 8.00-9.00 sec 438 MBytes 3.67 Gbits/sec 0 3.06 MBytes
[ 5] 9.00-10.00 sec 442 MBytes 3.72 Gbits/sec 0 3.06 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval Transfer Bitrate Retr
[ 5] 0.00-10.00 sec 4.23 GBytes 3.64 Gbits/sec 0 sender
[ 5] 0.00-10.00 sec 4.23 GBytes 3.64 Gbits/sec receiver
```

The resulting throughput when only the OpenWrt firewall is enabled is 3.64 Gbit/s.

Third case: only internal firewall enabled

```
/ # iperf3 -c 192.168.16.1
Connecting to host 192.168.16.1, port 5201
[ 5] local 192.168.16.60 port 58266 connected to 192.168.16.1 port 5201
[ ID] Interval Transfer Bitrate Retr Cwnd
[ 5] 0.00-1.00 sec 448 MBytes 3.76 Gbits/sec 0 553 KBytes
[ 5] 1.00-2.00 sec 448 MBytes 3.76 Gbits/sec 0 1.45 MBytes
[ 5] 2.00-3.00 sec 458 MBytes 3.83 Gbits/sec 0 2.04 MBytes
[ 5] 3.00-4.00 sec 455 MBytes 3.83 Gbits/sec 0 2.61 MBytes
[ 5] 4.00-5.00 sec 456 MBytes 3.83 Gbits/sec 0 3.02 MBytes
[ 5] 5.00-6.00 sec 460 MBytes 3.86 Gbits/sec 0 3.02 MBytes
[ 5] 6.00-7.00 sec 454 MBytes 3.81 Gbits/sec 0 3.02 MBytes
[ 5] 7.00-8.00 sec 448 MBytes 3.75 Gbits/sec 0 3.02 MBytes
[ 5] 8.00-9.00 sec 458 MBytes 3.84 Gbits/sec 0 3.02 MBytes
[ 5] 9.00-10.00 sec 430 MBytes 3.61 Gbits/sec 0 3.02 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval Transfer Bitrate Retr
[ 5] 0.00-10.00 sec 4.41 GBytes 3.79 Gbits/sec 0 sender
[ 5] 0.00-10.00 sec 4.41 GBytes 3.79 Gbits/sec receiver
```

The resulting throughput when only the internal firewall is enabled is 3.79 Gbit/s.

Fourth case: both firewalls disabled

```
/ # iperf3 -c 192.168.16.1
Connecting to host 192.168.16.1, port 5201
[ 5] local 192.168.16.60 port 58254 connected to 192.168.16.1 port 5201
[ ID] Interval Transfer Bitrate Retr Cwnd
[ 5] 0.00-1.00 sec 457 MBytes 3.83 Gbits/sec 0 1.01 MBytes
[ 5] 1.00-2.00 sec 466 MBytes 3.91 Gbits/sec 0 2.02 MBytes
[ 5] 2.00-3.00 sec 469 MBytes 3.94 Gbits/sec 0 2.59 MBytes
[ 5] 3.00-4.00 sec 465 MBytes 3.89 Gbits/sec 0 2.85 MBytes
[ 5] 4.00-5.00 sec 464 MBytes 3.89 Gbits/sec 0 3.15 MBytes
[ 5] 5.00-6.00 sec 464 MBytes 3.90 Gbits/sec 0 3.15 MBytes
[ 5] 6.00-7.00 sec 461 MBytes 3.87 Gbits/sec 0 3.15 MBytes
[ 5] 7.00-8.00 sec 460 MBytes 3.85 Gbits/sec 0 3.15 MBytes
[ 5] 8.00-9.00 sec 458 MBytes 3.85 Gbits/sec 0 3.15 MBytes
[ 5] 9.00-10.00 sec 470 MBytes 3.94 Gbits/sec 0 3.15 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval Transfer Bitrate Retr
[ 5] 0.00-10.00 sec 4.52 GBytes 3.89 Gbits/sec 0 sender
[ 5] 0.00-10.00 sec 4.52 GBytes 3.89 Gbits/sec receiver
```

The resulting throughput when both the firewalls are disabled is 3.89 Gbit/s.

From the analysis carried out it can be seen that firewalls have an impact on network throughput. In particular, starting from the configuration in which both are disabled, the internal firewall has a negative impact of about 2.6%, while the OpenWrt firewall decreases throughput by about 6.4%.Overall, considering both firewalls active, the throughput is decreased by about 9%. Consider that these measurements are not completely accurate as throughput varies slightly over time.

6.5 Detect port scans with Snort

Considering an attack aimed to penetrate inside a network, one of the early phases is the port scanning. In this phase the attacker knows the IP of the victim machine and performs a scan of all the ports to find active services and collect as much information as possible about that machine. This section analyzes how to detect those scans with the IDS implemented inside the container.

The most used tool for port scanning is "nmap" which is an open source utility for network discovery and security auditing. The default and most adopted option for port scanning is called "TCP SYN scan". It is really fast (thousands of ports scanned per second), relatively hidden and minimally invasive, as it never completes TCP connections.

To detect this kind of scan the IDS needs to be properly configured. Snort provides a module named "sfportscan" which has many options such as memory to save packets and analyze them based on time out and number of connections. To enable it open the "snort.conf" file, find the line `# preprocessor sfportscan:...` under the section `5)Configure preprocessors` and change it with the following line:

```
preprocessor sfportscan: proto { all } scan_type { all } sense_level { high }
logfile { scan }
```

that enables the "sfportscan" preprocessor. This operation allows the IDS to detect SYN and other types of scan and to log them in the log directory in the "scan" file declared in the option `logfile`.

Also another configuration can be performed to reinforce the scan detection. The "sfportscan" module gathers and logs information about the overall scan but, as it is shown later in the test, does not allow to see the individual packets intercepted. To perform this operation instead a rule needs to be created:

```
alert tcp any any -> any any (flags:S; msg:"TCP SYN packet intercepted"; sid:
1000005;)
```

This rule matches the TCP packets with the SYN flag set thanks to the option `flags:S` and needs to be inserted in a rules file included in the "snort.conf".

Now the scan detection is fully enabled and ready to be tested. All these operations have to be done before building the image in order to have the configuration already enabled when the container starts.

testing

After building the images with the script "build.sh" to integrate the new configuration, start the application running the "start.sh" script. At this point to properly test the IDS container two terminals should be connected to it. One to monitor in real time the intercepted packets and one to check the log files. The first terminal can be connected with the command `sudo docker attach ids` while the second should run the command `sudo docker exec -it ids sh` to open and attach a new shell in the container. Note that the same procedure can be performed also by executing the commands from the host with the Docker API `docker exec CONTAINER COMMAND`. Then to enable the IDS the command `snort -A console -q -c /etc/snort/snort.conf -i veth5b` is executed in the first terminal. At this point, from a device outside the application

network, the scan with "nmap" is executed running the command `nmap <router-external-IP>`. The scan in the test has been performed from a device with IP 192.168.1.5 to the external router IP 192.168.1.79 with the command `nmap 192.168.1.79`. This command executes a "TCP SYN scan" on the machine targeted. All the thousands of TCP packets intercepted by the IDS are shown in real time in the first terminal. One line is reported as an example:

```
02/07-18:44:53.030339 [**] [1:1000005:0] TCP SYN packet intercepted [**] [Priority:
0] TCP 192.168.1.5:60957 -> 192.168.1.79:1055
```

The line shows the time, the IP and port of the sender and recipient. Each line signals a packet intercepted.

When the scan terminates the stream of packets on the screen also ends. At this point on the other terminal connected to the container it's possible to see the log files without disabling the IDS. In the directory `/var/log/snort/` there should be at least two files. One is called "scan" which is a standard text file created by the "sfportscan" module and has the following content:

```
Time: 02/07-18:44:14.840594
event_ref: 0
192.168.1.5 -> 192.168.1.79 (portscan) TCP Filtered Portscan
Priority Count: 0
Connection Count: 200
IP Count: 1
Scanner IP Range: 192.168.1.5:192.168.1.5
Port/Proto Count: 200
Port/Proto Range: 21:60443
```

It shows the time the scan was started, the IP of the sender and the receiver, the type of scan and the range of ports scanned.

The other log file is called "snort.log.<timestamp>" where <timestamp> is the moment when Snort was started in IDS mode marked in Unix time. This file can be read with the command `snort -r snort.log.<timestamp>` and contains the details of all the individual packets intercepted as follows:

```
WARNING: No preprocessors configured for policy 0.
02/07-18:44:53.198436 192.168.1.5:60961 -> 192.168.1.79:14238
TCP TTL:254 TOS:0x0 ID:0 IpLen:20 DgmLen:64 DF
******S* Seq: 0xD36DD6E2 Ack: 0x0 Win: 0xFFFF TcpLen: 44
TCP Options (8) => MSS: 1460 NOP WS: 6 NOP NOP TS: 1903800874 0 SackOK EOL
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

WARNING: No preprocessors configured for policy 0.
02/07-18:44:53.240816 192.168.1.5:60962 -> 192.168.1.79:7019
TCP TTL:254 TOS:0x0 ID:0 IpLen:20 DgmLen:64 DF
******S* Seq: 0xDDACF152 Ack: 0x0 Win: 0xFFFF TcpLen: 44
TCP Options (8) => MSS: 1460 NOP WS: 6 NOP NOP TS: 1903800915 0 SackOK EOL
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Chapter 7

User manual

7.1 Overview

This guide explains how to start the application and use the services provided. The application creates a private network consisting of:

- A router and a perimeter firewall;
- An intrusion detection system (IDS);
- A switch and an internal firewall;
- A Firefox web browser;
- An access point that allows wireless connection to the network.

The application takes advantage of container technology using Docker container engine. Each component or pair of components listed is isolated in a different container that therefore provides a specific service. Those containers are interconnected as shown in the figure 7.1.

7.2 Requirements

This section illustrates the requirements to use the application.

7.2.1 System requirements

It's necessary to have a Linux based system provided with a wireless card and Internet connection through Ethernet. The application has been tested and developed on a computer with Ubuntu 20.04 LTS. The application has been developed for x86-64 architecture systems.

To install Linux follow this guide.

7.2.2 Software requirements

In order to run the application it's necessary to have Docker installed on the system. To install docker follow this guide.

Also it's necessary to have the "thesisproject" folder which contains all the application components.

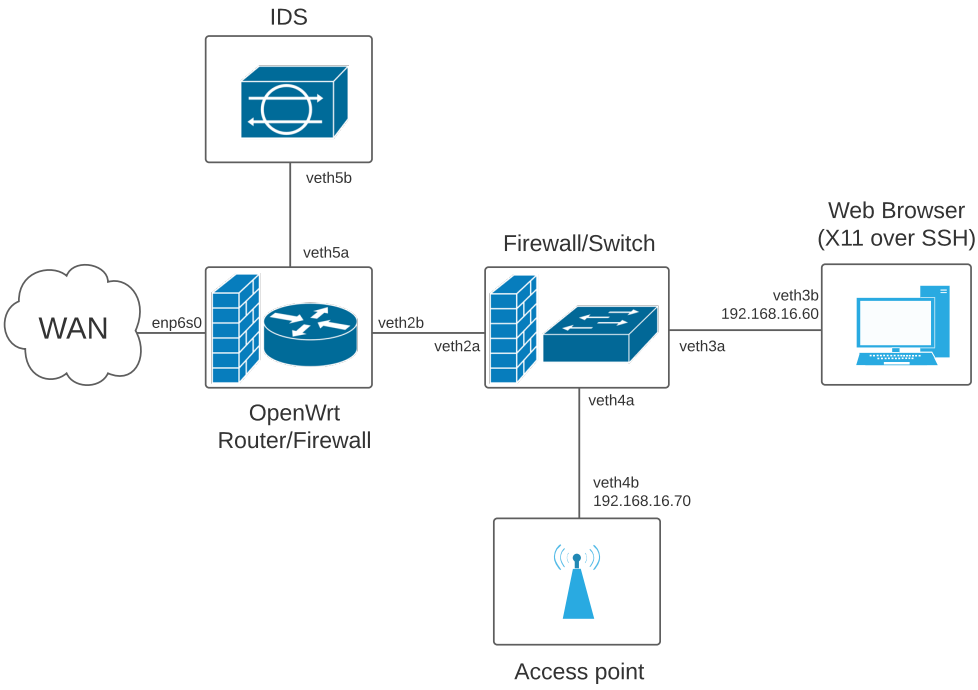

Figure 7.1. Application topology

7.3 Configuration

Some configurations must be performed before starting the application, otherwise it will start with the default values which may not be correct.

7.3.1 Change Wi-Fi name and password

It is possible to change the Wi-Fi name and password of the access point before starting the application. This operation is not mandatory but is highly recommended for security reasons. If it is not changed the default password will be "password".

To change the password, open the application "start.sh" script with any text editor. Find the following lines at the top of the script:

```
#set wi-fi name
SSID='apdocker_wpa2'

#set wi-fi password
WIFIPASSWORD='password'
```

The values inside the single quotes are assigned to the two variables. Change them if desired and save the file. Those values will be the new name and password of the Wi-Fi when the application is started.

7.3.2 Change OpenWrt container password for SSH

The OpenWrt container enables SSH connections on port 22 by default each time it is started. The password for this connections is set automatically by the script when the application starts

and is customizable. To change the password, open the application "start.sh" script with any text editor. Find the following lines at the top of the script:

```
#set OpenWrt container password (SSH)
OWPASS='password'
```

The value inside the single quotes will be the OpenWrt container password for SSH. Change it if desired and save the file.

7.3.3 Change wireless card interface name

The default wireless card name is "wlp7s0". If the name of the wireless card is different from the default values it is mandatory to change it in order to make the application work. To find out this information on the current Linux system open a terminal and run the command `iw dev`. This command lists all network interfaces for wireless hardware. The output should be similar to the following:

```
user@user:~/Desktop$ iw dev
phy#0
        Interface wlp7s0
                ifindex 3
                wdev 0x1
                addr b8:76:3f:4e:43:25
                ssid NETGEAR24
                type managed
                channel 10 (2457 MHz), width: 20 MHz, center1: 2457 MHz
                txpower 20.00 dBm
```

The line `Interface wlp7s0` shows the name of the wireless card, which in this case is "wlp7s0" as the default value. If the name is different the following operations must be performed.

Open the application "start.sh" script with any text editor. Find the following lines at the top of the script:

```
#set wireless card name
WC='wlp7s0'
```

The string inside the single quote must be the correct wireless card name. Change it if it's different from "wlp7s0" and save the file before starting the application.

7.3.4 Change Ethernet card interface name

The default Ethernet interface name is "enp6s0". This is the interface connected to the WAN which provides the Internet connection. If the name of the interface is different from the default values it is mandatory to change it in order to make the application work. To find out this information on the current Linux system open a terminal and run the command `ip link show`. This command lists all network interfaces on the host. The output should be similar to the following:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
    DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp6s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
    mode DEFAULT group default qlen 1000
    link/ether 74:d0:2b:46:c8:3b brd ff:ff:ff:ff:ff:ff
3: wlp7s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
    DORMANT group default qlen 1000
    link/ether b8:76:3f:4e:43:25 brd ff:ff:ff:ff:ff:ff
```

In this case the Ethernet interface is "enp6s0" which is the second in the list and may be different in other systems. If the name is different the following operations must be performed.

Open the application "start.sh" script with any text editor. Find the following lines at the top of the script:

```
#set WAN interface name
INT='enp6s0'
```

The string inside the single quote must be the correct Ethernet interface. Change it if it's different from "enp6s0" and save the file before starting the application.

7.4 Build the images

The application makes use of Docker images to start the containers. Those images needs to be built before starting the applications. To perform this operation open a terminal and move to the "thesisproject" folder, then run the "build.sh" script with the command:

```
sudo ./build.sh
```

The command requires administrator (root) permissions. Wait for the execution to finish.

7.5 Application startup

To execute the application open a terminal, move to the "thesisproject" directory and run the "start.sh" script with the command:

```
sudo ./start.sh
```

Administrator (root) permissions are needed to execute the program.

The following line should appear on the screen if the application started correctly:

```
user@user:~$ sudo ./start.sh
[sudo] password for user:
[+] Starting OpenWrt router container
d2afaa6b5b61e68c3dd859d94a4a06a2afeb1a01310b529f1018977a70b9e89a
[+] Container name: openwrt_1
[+] Starting switch/firewall container
5969cd8fd63b0f358c61bc31633f5e311a826365c249bdcb189e89ffe958db0a
[+] Container name: firewall
[+] Starting firefox container
62d6817a24470ec074482e6ebffe8822c93e8d80988c8d835209d5ea7dea2479
[+] Container name: firefox
[+] Container IP: 192.168.16.60
[+] Starting access point container
d355abe76387ae50814ffe6e1f1ab63e42c751c96bc41155075fa88d7f978958
[+] Container name: ap
[+] Container IP: 192.168.16.70
[+] Starting Snort IDS container
4099c0ad6df99e02a8488f73d6aab59795e2cd8c234bfb2a19fd189cb9f98309
[+] Container name: ids
[+] Creating interfaces to connect containers
[+] Connecting switch and router containers
[+] Connecting firefox container to switch
[+] Connecting access point container to switch
[+] Connecting ids container to router
[+] Configuring containers
[+] Moving wireless card interface wlp7s0 to access point namespace
```

```
[+] Enabling access point
[+] Changing OpenWrt container password
Changing password for root
New password:
Bad password: too weak
Retype password:
passwd: password for root changed by root
[+] Mirror traffic on router to ids
[+] Application started
[+] To attach container run: sudo docker attach <container-name>
```

The startup can take some seconds, wait until it ends. It's possible to display a list with all the active containers executing the command `sudo docker ps` to make sure that the containers are running.

The output on the screen gives important information about the application components. It shows all the containers name, the IP of those connected to the switch and lastly the command to enter in a container from the host terminal.

7.6 Terminate the application

To terminate the execution of the application open a terminal, move to the "thesisproject" directory and run the "stop.sh" script with the command:

```
sudo ./stop.sh
```

The name of the terminated containers is printed on the screen.

7.7 Application error

If any errors occur during application startup the last lines of the output on the screen will be similar to the following.

```
[+] Error occurred!
Command that caused the error: "docker exec openwrt_1 opkg update"
Cleaning up the following containers:
ids
ap
firefox
firewall
openwrt_1
[+] Exiting!
```

In this case a command failed during the execution of the script. The command that raised the error is reported on the screen for debug purposes.

7.8 Application usage

The names of the containers are specified in the screen output. In particular:

openwrt_1 is the OpenWrt router and perimeter firewall container;

firewall is the container with the switch and the internal firewall;

firefox is the web browser container;

ap is the access point container;

ids is the intrusion detection system container.

The output also specifies the private IP address of the containers connected to the switch.

7.8.1 Access the OpenWrt router container

The OpenWrt container can be accessed from the host with the command

```
sudo docker attach openwrt_1
```

It can also be accessed from any devices within the application LAN through SSH with the command

```
ssh root@192.168.16.1
```

then the password for the OpenWrt router container is requested to perform the access. For more information refer to 7.3.2.

7.8.2 Manage the container services

All the containers are accessible from the host for managing the network services. To access a container open a terminal and run the command:

```
sudo docker attach <container-name>
```

Then a shell inside the container should appear in the current terminal and the commands can be executed directly inside it. The names of the containers are shown in the output of the "start.sh" script or they can be retrieved by running the command `sudo docker ps` that shows a list of all the active containers and their names.

Another equivalent method to execute commands inside the containers from the host without accessing them is to use the Docker API `docker exec` as follows:

```
sudo docker exec <container_name> [COMMAND]
```

which is the same as executing `[COMMAND]` inside the container named "<container_name>".

7.8.3 Connect devices to the Wi-Fi

The access point creates a Wi-Fi network. Devices can be connected wireless to it. The access point name and password are the ones set following the section 7.3.1, which illustrates how to change the Wi-Fi password and SSID. If this operation has not been performed, the default password is "password" and the default SSID is "apdocker_wpa2". It is highly recommended to change those values as the default ones are insecure.

7.8.4 Use the Firefox web browser inside the container

Note: the web browser runs inside the container and its graphical interface is forwarded over SSH using a protocol called X11. X11 is a computer software and network protocol that provides a graphical user interface (GUI) for networked computers common on Unix-like systems. Therefore some systems support this functionality natively while others do not. To perform the operation illustrated in this section make sure that the system used supports X11. If not it's necessary to download a software that integrates this functionality in the current system.

The "firefox" container offers an isolated environment to perform potentially dangerous operations such as browsing dangerous Internet sites or downloading untrusted files. The container can be accessed using the SSH protocol from devices connected to the network, the container IP

is 192.168.16.60. To access the container open a terminal on a device connected to the network and execute the command:

```
ssh -Y user@192.168.16.60
```

Then insert the password. The default password is "password". To change user or password refer to 8.6.4.

Once the container is connected to open the web browser run the command `firefox`. The Firefox GUI should be prompted on the screen.

This container is mainly dedicated to execute a web browser but it can be used also to execute other X11 applications or other pieces of software. Software present as a package in the Alpine Linux packages library can be downloaded with the command `apk add <package-name>`.

To close the SSH connection run the command `exit` inside the SSH terminal.

7.8.5 Enable and use the IDS

The IDS is disabled by default and can be enabled at any time by attaching the container with the command `sudo docker attach ids` and then running the command:

```
snort -A console -c /etc/snort/snort.conf -i veth5b
```

This command starts the tool in IDS mode showing in real time the intercepted packets matching the chosen rules (see 8.6.3). The `-q` option (quiet) can be added to hide the verbose startup configuration output. The same operation can be also performed from the host terminal by running the same command preceded by `sudo docker exec ids`.

When Snort is active in IDS mode its operations are continuously logged. The default path to the log files is `/var/log/snort/`. There are two types of log file: the files that contain the information on the individual packets intercepted and the files that contain other general information. The first type are PCAP files, which is a specific format to record packet data from a network scan, and can be read with the command `snort -r <file-name>`. The others are standard text files.

The IDS is configured to detect network scans that target the application network, for more information refer to 6.5.

7.9 Update the images

To update the images, simply run the "build.sh" script as shown in section 7.4. The images will be rebuilt to the latest version.

If the application is running and the host has no Internet connection, before rebuilding the images it's necessary to connect the host to the LAN created by the application. To perform this operation move to the "thesisproject" directory and run the "connecthost.sh" script with the command:

```
sudo ./connecthost.sh
```

It is strongly recommended to read section 8.4.5 before executing this script as some modifications may have to be performed.

After rebuilding the images, the application must be terminated and restarted to integrate the new updates. See 7.6 (stop) and 7.5 (start).

Chapter 8

Developer manual

8.1 Introduction

The application creates a private network composed of a router, an IDS, a switch, a perimeter firewall and an internal firewall, a web browser and an access point.

The application takes advantage of container technology using Docker container engine. Each component or pair of components is isolated in a different container that therefore provides a specific service. Those containers are interconnected as shown in figure 8.1.

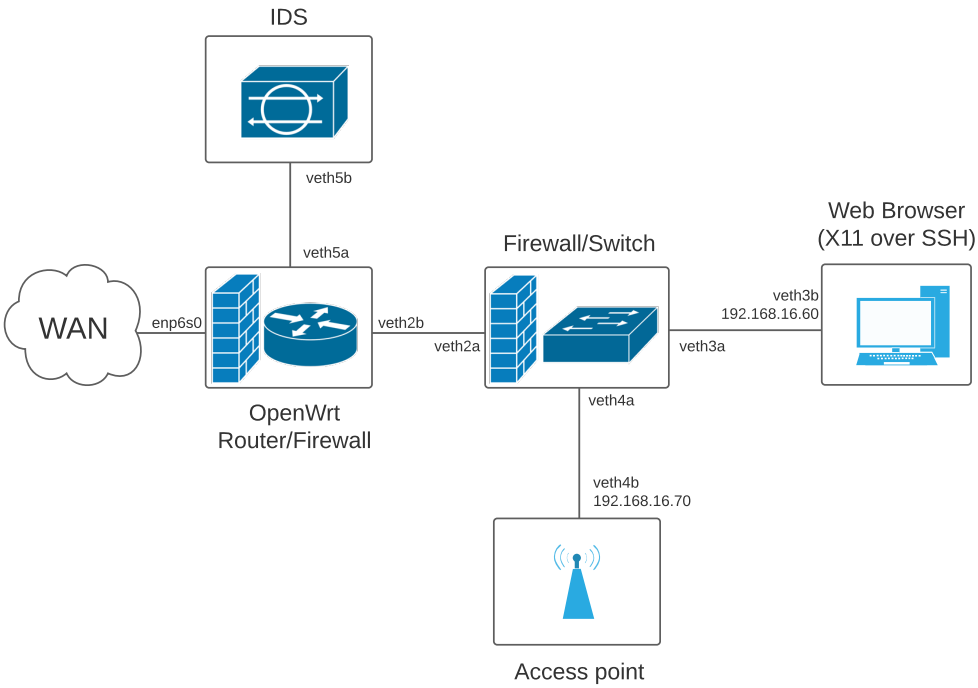

Figure 8.1. Application topology

8.2 Images and software

To create the containers, the application needs the related Docker images. Those images must be present on the host when the application is executed. The main software used to build the images are:

- Openwrt for the router and perimeter firewall;
- Snort as the intrusion detection system (IDS);
- Linux kernel features: Iptables for the internal firewall and Linux bridges for the switch;
- Firefox, OpenSSH and X11 for the web browser image;
- Hostapd, dhcp, iptables for the access point.

8.3 Application components

All the application components are located in the "thesisproject" folder. The folder has the following content:

```
accesspoint build.sh connecthost.sh ids router start.sh stop.sh
    switchfirewall webbrowser

./accesspoint:
confs Dockerfile

./accesspoint/confs:
dhcpd.conf hostapd_confs iptables_off.sh iptables.sh start.sh

./accesspoint/confs/hostapd_confs:
accept wpa2.conf

./ids:
all_rules config Dockerfile rules

./ids/all_rules:
rules

./ids/all_rules/rules:
(list of all the snort rules files)

./ids/config:
classification.config reference.config snort.conf threshold.conf unicode.map

./ids/rules:
black_list.rules white_list.rules

./router:
Dockerfile dropbear network

./router/dropbear:
dropbear_ed25519_host_key dropbear_rsa_host_key

./switchfirewall:
Dockerfile enablefirewall.sh
```

```
./webbrowser:
Dockerfile ssh

./webbrowser/ssh:
moduli sshd_config ssh_host_dsa_key.pub ssh_host_ecdsa_key.pub
    ssh_host_ed25519_key.pub ssh_host_rsa_key.pub
ssh_config ssh_host_dsa_key ssh_host_ecdsa_key ssh_host_ed25519_key
    ssh_host_rsa_key
```

The folder contains four scripts to manage the application together with five folders with the material to build the related five images. The four scripts are analyzed in depth in the next section.

8.4 Application scripts

The application consists of four bash scripts: "build.sh", "start.sh", "stop.sh", "connecthost.sh". Before explaining these scripts it is best to illustrate all the APIs and functions that they use.

8.4.1 Functions and APIs

The scripts use APIs provided by Docker to interact with the Docker daemon, which manages the containers. The APIs used are `docker build [OPTIONS] PATH` to build Docker images, `docker inspect [OPTIONS] NAMEID [NAMEID...]` to find out container informations, `docker run [OPTIONS] IMAGE[:TAG@DIGEST] [COMMAND] [ARG...]` to run a container and `docker exec [OPTIONS] CONTAINER COMMAND [ARG...]` to execute commands inside a running container.

Finally to stop the container execution `docker rm [OPTIONS] CONTAINER [CONTAINER...]`.

The "start.sh" script makes great use of the `ip` Linux command which is used for performing several network administration tasks. In particular it is used to create and manipulate veth interfaces for connecting containers together. Those interfaces are created on the host and then moved to the containers namespaces to connect them. Two containers are connected if they have the two endpoints of a veth inside their namespaces.

8.4.2 Build script

The "build.sh" script builds the five images that will be used to start the application. The content is:

```
#!/bin/bash

cd router
docker build --pull . -t router:thesis
cd ../switchfirewall
docker build --pull . -t switchfirewall:thesis
cd ../webbrowser
docker build --pull . -t firefoxalpine:thesis
cd ../ids
docker build --pull . -t ids:thesis
cd ../accesspoint
docker build --pull . -t accesspoint:thesis
```

The script changes one by one the five directories building for each one the relative image. The command `docker build --pull . -t <image-name>:<tag>` builds an image based on the Dockerfile present in the directory with the name and tag specified. The option `--pull` makes sure that the base image in the Dockerfile is updated by pulling it from the DockerHub. All the images are tagged "thesis" and are used in the other two application scripts.

8.4.3 Start script

The "start.sh" script is the core of the application. It starts all the containers, configures and connects them. The content is:

```
1  #!/bin/bash
2
3  #set wi-fi password
4  WIFIPASSWORD='password'
5
6  #set wi-fi name
7  SSID='apdocker_wpa2'
8
9  #set OpenWrt container password (SSH)
10 OWPASS='password'
11
12 #set wireless card name
13 WC='wlp7s0'
14
15 #set WAN Ethernet interface name
16 INT='enp6s0'
17
18 NOCOLOR='\033[0m'
19 GREEN='\033[0;32m'
20 CYAN='\033[0;36m'
21 RED='\033[0;31m'
22
23 cleanup(){
24     echo -e "${RED}[+] Error occurred! \nCommand that caused the error:
          \"${last_command}\" \nCleaning up the following containers:
          ${NOCOLOR}"
25     docker rm -f ids ap firefox firewall openwrt_1
26     echo -e "${RED}[+] Exiting! ${NOCOLOR}"
27     exit 0
28 }
29
30 #call cleanup function when any command fails
31 trap 'cleanup' ERR
32 #keep track of the last executed command
33 trap 'last_command=$current_command; current_command=$BASH_COMMAND' DEBUG
34
35 echo -e "${GREEN}[+] Starting OpenWrt router container ${NOCOLOR}"
36 docker run --cap-add=NET_ADMIN -dit --name openwrt_1 --network none
       --memory="500m" --memory-swap="500m" --cpus="0.5" -v
       $(pwd)/router/dropbear:/etc/dropbear:ro router:thesis
37 echo -e "${CYAN}[+] Container name: openwrt_1 ${NOCOLOR}"
38
39 docker exec openwrt_1 sed -i "s/enp6s0/$INT/g" /etc/config/network
40
41 echo -e "${GREEN}[+] Starting switch/firewall container ${NOCOLOR}"
42 docker run --cap-add NET_ADMIN --name firewall --net=none -itd
       --memory="100m" --memory-swap="100m" --cpus="0.2" switchfirewall:thesis
43 echo -e "${CYAN}[+] Container name: firewall ${NOCOLOR}"
44
45 echo -e "${GREEN}[+] Starting firefox container ${NOCOLOR}"
46 docker run --name firefox --net=none -itd --add-host firefox:192.168.16.60
       --hostname firefox --dns 8.8.8.8 --dns 8.8.4.4 --memory="1200m"
       --memory-swap="1200m" --cpus="1.0" firefoxalpine:thesis
```

```
47 echo -e "${CYAN}[+] Container name: firefox ${NOCOLOR}"
48 echo -e "${CYAN}[+] Container IP: 192.168.16.60 ${NOCOLOR}"
49
50 echo -e "${GREEN}[+] Starting access point container ${NOCOLOR}"
51 docker run --name ap --cap-add=NET_ADMIN --network=none -dit --memory="300m"
       --memory-swap="300m" --cpus="0.5" accesspoint:thesis
52 echo -e "${CYAN}[+] Container name: ap ${NOCOLOR}"
53 echo -e "${CYAN}[+] Container IP: 192.168.16.70 ${NOCOLOR}"
54
55 echo -e "${GREEN}[+] Starting Snort IDS container ${NOCOLOR}"
56 docker run --name ids -itd --net=none --cap-add=NET_ADMIN --memory="1200m"
       --memory-swap="1200m" --cpus="0.8" ids:thesis
57 echo -e "${CYAN}[+] Container name: ids ${NOCOLOR}"
58
59 echo -e "${GREEN}[+] Creating interfaces to connect containers ${NOCOLOR}"
60 ip link add veth2a type veth peer name veth2b
61 ip link set up veth2a
62 ip link set up veth2b
63
64 ip link add veth3a type veth peer name veth3b
65 ip link set up veth3a
66 ip link set up veth3b
67
68 ip link add veth4a type veth peer name veth4b
69 ip link set up veth4a
70 ip link set up veth4b
71
72 ip link add veth5a type veth peer name veth5b
73 ip link set up veth5a
74 ip link set up veth5b
75
76 #create bridge inside switch/firewall container
77 docker exec firewall ip link add name br type bridge
78 docker exec firewall ip link set br up
79
80 echo -e "${GREEN}[+] Connecting switch and router containers ${NOCOLOR}"
81 owpid=$(docker inspect -f '{{.State.Pid}}' openwrt_1)
82 ln -sfT /proc/${owpid}/ns/net /var/run/netns/openwrt_1
83 ip link set veth2b netns openwrt_1
84 ip netns exec openwrt_1 ip link set up veth2b
85
86 #move WAN Ethernet interface to router namespace
87 ip link set $INT netns openwrt_1
88 ip netns exec openwrt_1 ip link set up $INT
89
90 fwpid=$(docker inspect -f '{{.State.Pid}}' firewall)
91 mkdir -p /var/run/netns/
92 ln -sfT /proc/${fwpid}/ns/net /var/run/netns/firewall
93 ip link set veth2a netns firewall
94 ip netns exec firewall ip link set up veth2a
95
96 docker exec firewall ip link set veth2a master br
97
98 echo -e "${GREEN}[+] Connecting firefox container to switch ${NOCOLOR}"
99 ffpid=$(docker inspect -f '{{.State.Pid}}' firefox)
100 ln -sfT /proc/${ffpid}/ns/net /var/run/netns/firefox
101 ip link set veth3b netns firefox
```

```
102 ip netns exec firefox ip link set up veth3b
103 ip netns exec firefox ip addr add 192.168.16.60/16 dev veth3b
104 ip netns exec firefox ip route add default via 192.168.16.1
105
106 ip link set veth3a netns firewall
107 ip netns exec firewall ip link set up veth3a
108
109 docker exec firewall ip link set veth3a master br
110
111 echo -e "${GREEN}[+] Connecting access point container to switch ${NOCOLOR}"
112 appid=$(docker inspect -f '{{.State.Pid}}' ap)
113 ln -sfT /proc/${appid}/ns/net /var/run/netns/ap
114 ip link set veth4b netns ap
115 ip netns exec ap ip link set up veth4b
116 ip netns exec ap ip addr add 192.168.16.70/16 dev veth4b
117 ip netns exec ap ip route add default via 192.168.16.1
118
119 ip link set veth4a netns firewall
120 ip netns exec firewall ip link set up veth4a
121 docker exec firewall ip link set veth4a master br
122
123 echo -e "${GREEN}[+] Connecting ids container to router ${NOCOLOR}"
124 idspid=$(docker inspect -f '{{.State.Pid}}' ids)
125 ln -sfT /proc/${idspid}/ns/net /var/run/netns/ids
126 ip link set veth5b netns ids
127 ip netns exec ids ip link set up veth5b
128
129 ip link set veth5a netns openwrt_1
130 ip netns exec openwrt_1 ip link set up veth5a
131
132 echo -e "${GREEN}[+] Configuring containers ${NOCOLOR}"
133
134 #change wi-fi name and password inside ap container
135 docker exec ap sed -i "s/apdocker_wpa2/$SSID/g" /etc/hostapd/hostapd.conf
136 docker exec ap sed -i "s/password/$WIFIPASSWORD/g" /etc/hostapd/hostapd.conf
137
138 #change wireless card name inside ap container
139 docker exec ap sed -i "s/wlp7s0/$WC/g" /etc/hostapd/hostapd.conf /iptables.sh
        /iptables_off.sh /start.sh
140
141 echo -e "${GREEN}[+] Moving wireless card interface ${WC} to access point
        namespace ${NOCOLOR}"
142 #move wireless card to ap namespaces
143 PHY=$(cat /sys/class/net/$WC/phy80211/name)
144 iw phy $PHY set netns $appid
145 docker exec ap ip link set $WC up
146 docker exec ap ip addr add 11.0.0.1/24 dev $WC
147
148 echo -e "${GREEN}[+] Enabling access point ${NOCOLOR}"
149 docker exec -d ap ./start.sh
150
151 echo -e "${GREEN}[+] Changing OpenWrt container password ${NOCOLOR}"
152 (echo "$OWPASS"; echo "$OWPASS") | docker exec -i openwrt_1 passwd
153
154 echo -e "${GREEN}[+] Mirror traffic on router to ids ${NOCOLOR}"
155 docker exec openwrt_1 nft add table netdev filter
```

```
156 docker exec openwrt_1 nft add chain netdev filter ingress0 '{ type filter
        hook ingress devices = { '"${INT}"', br-lan } priority 0; }'
157 docker exec openwrt_1 nft add rule netdev filter ingress0 dup to veth5a
158
159 echo -e "${GREEN}[+] Application started ${NOCOLOR}"
160 echo -e "${GREEN}[+] To attach contaner run: sudo docker attach
        <container-name> ${NOCOLOR}"
```

The script can be divided into five sections to facilitate the analysis:

Lines 3 to 21 are dedicated to variables initialization;

Lines 23 to 33 deal with possible script errors;

Lines 35 to 57 start the five containers from the relative images;

Lines 59 to 130 deal with the connection of the containers;

Lines 132 to 160 are dedicated to the containers configuration.

Those sections are analyzed one by one in the following paragraphs.

Variables initialization (lines 3-21)

In these lines all the variables used throughout the script are initialized. They should be changed according to the user needs before starting the application. The variables are:

WIFIPASSWORD defines the Wi-Fi password;

SSID defines the Wi-Fi network name;

OWPASS defines the OpenWrt container password for SSH;

WC defines the wireless card name;

INT defines the Ethernet interface name;

Other variables are just colors definition used in the `echo` commands to improve the visibility of the script output.

Error handling (lines 23-33)

It's good practice in every script to handle possible command failures. If a command fails it is better to find out the problem and solve it than to risk that the application breaks without knowing why. Also if something fails and the script terminates it is good practice to perform a cleaning of all the components created before the failure. All these functionalities are implemented in this section.

The first lines implement the "cleanup" function which prints on the screen that an error has occurred and also which command raised it (with the variable `last_command` explained later). Then it removes the containers that may already be active to bring back the system to the initial state and finally terminates the script.

The command `trap 'cleanup' ERR` calls the function "cleanup" if any error occurs during the script execution which means if any function returns a non-zero exit value indicating a failure.

Finally the command

```
trap 'last_command=$current_command; current_command=$BASH_COMMAND' DEBUG
```

keeps track of the last command executed which will be printed in case it fails. The variables `last_command` and `current_command` contain respectively the last and the current command being executed.

Starting containers (lines 35-57)

This section of code starts the five containers with the Docker command:

`docker run [OPTIONS] IMAGE[:TAG@DIGEST] [COMMAND] [ARG...]`

The following containers are started from the specified images.

- The router container from the "router:thesis" image;
- The switch and firewall container from the "switchfirewall:thesis" image;
- The web browser container from the "firefoxalpine:thesis" image;
- The access point container from the "accesspoint:thesis" image;
- The IDS container from the "ids:thesis" image.

Each container is given a different name and provides a different service and therefore is started with different options. All the containers are started without any network connection (`--net=none`) because they will be interconnected later in the script. The option `--cap-add NET_ADMIN` is specified on the containers that need to perform network related tasks. All the containers are started with the option `-dit` which runs them interactively in the background allowing users to connect to them at a later time. The router and web browser containers are started with the option `-v` to mount the SSH folders with the correct host keys from the host to the containers.

Each container is started with a specific hardware resources limitation in order to avoid cases of resource starvation or, in the worst case, DoS attacks. The options used are

- `--memory="<value-MB>m"`
- `--memory-swap="<value-MB>m"`
- `--cpus="<core-value>"`

to respectively limit the RAM, the memory swap and the CPU. These values can be changed according to the system resources and to the services provided by the various containers.

The command at line 46 runs the "firefox" container with some additional options. The flag `add-host` adds the host "firefox" with IP 192.168.16.60 and then `hostname` gives the hostname "firefox" to the actual container. This is necessary to make the X11 screen forwarding work correctly. Finally the `dns` option adds the Google DNS servers IP to the container;

At line 39 the Ethernet interface stored in the variable "INT" is set inside the router container, specifically in the network configuration file.

Connecting containers (lines 59-130)

In this piece of code the containers are connected following the topology shown in figure 8.1. The connection between two containers is performed by creating a veth device with the command `ip link add <end-point1> type veth peer name <end-point2>`. The veth device created has two interfaces "<end-point1>" and "<end-point2>". The two containers are connected when the two endpoints are moved inside their namespaces with the command `ip link set <end-point> netns <container-name>` performed on both the containers.

At line 77 a bridge device is created inside the switch/firewall container. This device works as a switch and all the veth endpoints moved inside this container are connected to it allowing the other containers to join the network.

Also the containers connected to the switch are assigned an IP address and default route after the veth endpoints are moved inside their namespaces with the commands:

```
ip netns exec firefox ip addr add 192.168.16.60/16 dev veth3b
ip netns exec firefox ip route add default via 192.168.16.1
```

At line 87 the WAN Ethernet interface is moved from the default host namespace to the router container namespace in order to directly connect the router to the WAN.

Configuring containers (lines 132-160)

In the last section the containers are configured. The operations performed are:

- The correct Wi-Fi SSID, password, and wireless card name that are stored in the variables at the beginning of the script are set inside the access point container (lines 135 to 139);
- The wireless card interface is moved to the access-point container (lines 143 to 146);
- The access point is enabled (line 149);
- The OpenWrt password stored in the variable "OWPASS" is set inside the container (line 152);
- All the traffic in transit through the router is mirrored to the IDS container. To realize this procedure the traffic incoming and outgoing on the router is mirrored with nftables commands (lines 155 to 157). Nftables is the packet filtering framework of Openwrt.

8.4.4 Stop script

The "stop.sh" script terminates the execution of the application. The content is:

```
#!/bin/bash

#remove the container to stop the application
docker rm -f ids ap firefox firewall openwrt_1
```

The content it's just a single command that stops and removes all the containers in use by the application.

8.4.5 Connect host script

When the Ethernet and wireless card interfaces are moved to the containers namespaces, the host remains disconnected from any network, including Internet. Anyway, if needed, it can be connected to the LAN created by the application running the "connecthost.sh" script. The content of this script is the following:

```
#!/bin/bash

#connect host to switch
ip link add veth6a type veth peer name veth6b
ip addr add 192.168.16.90/16 dev veth6b
ip link set veth6a netns firewall
ip netns exec firewall ip link set up veth6a
ip netns exec firewall ip link set veth6a master br

#set DNS
printf "nameserver 8.8.8.8\nnameserver 8.8.4.4\n" >> /etc/resolv.conf

#add firewall rule
docker exec firewall iptables -I FORWARD 1 -m physdev --physdev-in veth2a
    --physdev-out veth6a -m state --state ESTABLISHED,RELATED -j ACCEPT
```

```
#set default route
ip route add default via 192.168.16.1
```

The script connects the default host namespace to the switch using a veth device, in the same way as a container would be connected. One veth endpoint is left in the host namespace and a static IP address is assigned to it. The other endpoint is moved in the switch/firewall container and added to the bridge. Then the Google DNS servers are added to the "resolv.conf" file of the host to make sure that the DNS service works correctly. At this point a rule that enables the host connection is added to the firewall. Finally the default route is set on the host.

The correct operation of this script is highly dependent on the configuration of the host which may vary from system to system. Consequently, the file may need to be modified according to the situation. For example, some commands may interfere with automatic network services on the system or the host may not need the Google DNS service. Anyway, even if modifications are made or some commands are executed manually, all the operations written in the script must be performed to enable the connection.

8.5 Network details

The network created is a LAN on the 192.168.16.0/16 subnet. The router gateway is at the IP 192.168.16.1.

8.5.1 Router

The router connects the LAN to the WAN. The WAN is the external network that provides Internet connection. The connection is performed by moving the Ethernet interface which is connected to the WAN inside the router container namespace (lines 87,88 of the "start.sh" script). The router is configured to get the IP of the Ethernet interface through DHCP from the WAN and to create the LAN.

8.5.2 Switch

To join the network a host needs to be connected to the switch. All the containers that provide network services are connected to the switch. It is built using a Linux bridge called "br" inside the switch/firewall container (lines 74,75). To connect a container to the switch the veth interface moved inside the switch/firewall container must be added to the bridge as follows:

```
sudo docker exec <switch-container> ip link set <veth-endpoint> master br
```

In addition, the internal firewall configuration should be updated accordingly.

8.6 Images configuration

The "build.sh" script builds the five images starting from the contents of the five folders in the "thesisproject" directory. These folders contain all the files to configure the services packaged inside the images and, although they have already been set during the development of the application, there are some files and concepts that the user or developer should be aware of.

8.6.1 Internal firewall configuration file

The file "enablefirewall.sh" inside the "switchfirewall" folder contains all the iptables rules to configure the internal firewall. The developer should edit this file if he wants to change the firewall rules for any reason.

8.6.2 OpenWrt perimeter firewall configuration file

OpenWrt offers a default firewall that creates a barrier between the internal private network and the external public network to which it is connected. The configuration file for this firewall is located at path `/etc/config/firewall` within the OpenWrt container and can be modified if necessary (see 4.2.2). The firewall can also be modified in a user-friendly way using the LuCi graphical interface offered by OpenWrt, which needs to be installed as it is not present by default (see 8.8.1). The described operations require the application to be running. To perform a permanent configuration that persists each time the container is started, it is necessary to change the configuration file in the Dockerfile to include the changes directly in the image.

8.6.3 IDS configuration file

The IDS configuration file is "snort.conf" under the path `thesisproject/ids/config`. Among all the settings this file contains the list of all the rules files used by the tool to identify the malicious network activity. Those rules can either be created for custom checks or downloaded from the snort site. In both cases to enable a set of rules they should be written inside a `.rules` file and moved to the `thesisproject/ids/all_rules/rules` directory which contains all the rules files. Then the new rules file should be included inside the file "snort.conf" under the section `7)Customize your rule set`

8.6.4 Web browser container SSH configuration

The credentials to access the "firefox" container through SSH are the same as the relative user account inside the container. The default user and password are respectively "user" and "password". Those values can be changed inside the relative Dockerfile at `thesisproject/webbrowser/`. The lines that configure user and password are:

```
RUN adduser user -D
RUN echo "user:password" | chpasswd
```

and, to modify user and password, can be changed as follows:

```
RUN adduser <new_user> -D
RUN echo "<new_user>:<new_password>" | chpasswd
```

Note that to create an SSH connection to the "firefox" container, a device must be already connected to the application LAN.

8.6.5 Network interfaces

The application uses two network interfaces: the one related to the wireless card and the one related to the Ethernet physical connection to the WAN. The default interfaces set in the images are "wlp7s0" and "enp6s0" which are the ones used during development. Even if the system on which the application is executed has different interfaces names, those inside the images configuration files should not be changed. Instead they should be left ad placeholders and the correct names should be configured in the "start.sh" file (see 8.4.3). The correct configuration is then performed at application runtime.

Note that if the interfaces names were correctly changed inside the images configuration files, the application would work anyway and the relative variables would become useless. However this operation is more complicated in terms of configuration and less functional in terms of portability.

8.6.6 Access point white-list and black-list access

White-list

It's possible to configure the access-point to allow only the devices with a specific MAC address to join the wireless network. The white-list file that contains the list of the MAC addresses (one per line) allowed to access the Wi-Fi is named "access" and is located at the path `thesisproject/accesspoint/confs/hostapd_confs/` (create the file if it doesn't exist). Also to enable this functionality the file "wpa2.conf" in the same path must contain the following variables definition:

```
macaddr_acl=1
accept_mac_file=/etc/hostapd/accept
```

Black-list

It's possible to deny the access to the network to devices with a specific MAC address. The black-list file that contains the list of the MAC addresses (one per line) to which the access is denied is named "deny" and is located at the path `thesisproject/accesspoint/confs/hostapd_confs/` (create the file if it doesn't exist). Also to enable this functionality the file "wpa2.conf" in the same path must contain the following variables definition:

```
macaddr_acl=0
deny_mac_file=/etc/hostapd/denied
```

Default access

To leave the default password authentication access to every device without implementing white-listing nor black-listing the "wpa2.conf" must contain the "macaddr_acl" variable set to 0 and no file ("deny", "accept") must be included. This option is adopted by default in the application.

8.6.7 Configure access point DHCP IP addresses range

The default access point DHCP configuration provides a range of 10 IP addresses (from 11.0.0.10 to 11.0.0.20), meaning that a maximum of 10 devices can connect to the wireless network. To change this configuration open the "dhcpd.conf" file at path `thesisproject/accesspoint/confs` and edit the line:

```
range 11.0.0.10 11.0.0.20;
```

with the desired IP addresses range on the subnet 11.0.0.0 with netmask 255.255.255.0.

8.7 Add new containers

It's possible to modify the base script to add functionalities and services to the application, for example by adding a new container to the network. To perform this operation, create a new Docker image that provides the desired service, then add to the script the new code to run the container based on that image and connect it to the switch following the same process adopted for the other containers. The container should be started with suitable options.

8.8 Download OpenWrt packages

It's possible to download packages to the Openwrt container to add functionalities if desired. The command to download the new packages from the host is:

```
sudo docker exec openwrt_1 opkg install [PACKAGE]
```

The command that follows `sudo docker exec` is executed inside the container. In case the command line interface is already connected to the container, this first part of the command needs to be removed. `opkg` is the packet manager of Openwrt.

This command can either be executed manually or inserted into the "start.sh" script under the configuration section to automate the download.

8.8.1 OpenWrt GUI

OpenWrt offers a functional web GUI called "LuCI" which is not present by default and needs to be downloaded. This graphical interface is not necessary but facilitates the management and configuration of the router and other software inside the container. This feature can be enabled by downloading two packages:

```
opkg install luci
opkg install luci-ssl
```

The second package allows the connection to the "LuCI" web interface with HTTPS support. Then from a device connected to the application LAN open a web browser and navigate to the secure URL `https://192.168.16.1` and the interface should appear asking for authentication. The standard username is "root" and the password required is the same password used to access OpenWrt via SSH. For more information refer to 7.3.2.

The process of downloading the packages can either be performed manually after the application startup or by adding the commands in the "start.sh" script under the configuration section.

TLS/SSL Certificate

The certificate used for the HTTPS connection is a self signed certificate automatically generated during the download of the "luci-ssl" package. For this reason the certificate will not be trusted by the web browser which will generate a warning. To deepen and resolve this issue and to generate a personal self signed certificate trusted by the web browser refer to https://openwrt.org/docs/guide-user/luci/getting_rid_of_luci_https_certificate_warnings.

Chapter 9

Conclusion

The application developed overcomes the shortcomings of some insecure gateways provided by ISPs or available on the market. It features a high level of service configuration and portability to different Linux-based devices. The application has been designed and tested to fit into limited resources systems through the choice of lightweight software that are free and easily available on the Internet, ensuring easy accessibility by users. The study conducted on the threats and risks related to container technology has allowed the development of the application with a dedicated attention to security. The document illustrates the whole development process of the single components and the related containers, allowing an easy analysis of the performed steps. The customization of the application is independent of the limitations that the ISP integrates into its products and provides a high level of security through the use of container technology and the integration of network security components such as IDS and firewall. The evaluation tests performed on the application provided an analysis of performance in several aspects. The adoption of Docker allows easy management of containers and images thanks to its large number of APIs. The application can be used in different environments and provides a starting point for new developments. It can be adapted to different contexts by extending its functionality or modifying its configuration in order to best suit each user's needs.

Bibliography

[1] M.Souppaya, J.Morello, K.Scarfone, "Application Container Security Guide", NIST Special Publication 800-190, September 2017, pp. 63, DOI 10.6028/NIST.SP.800-190

[2] K. Scarfone, W. Jansen, M. Tracy, "Guide to General Server Security", NIST Special Publication 800-123, July 2008, pp. 53, DOI 10.6028/NIST.SP.800-123

[3] Docker project documentation, https://docs.docker.com/get-started/overview/

[4] CloudSecDocs container security, https://cloudsecdocs.com/container_security/theory/threats/docker_threat_model/

[5] Automatic Access Point with Docker, https://fwhibbit.es/en/automatic-access-point-with-docker-and-raspberry-pi-zero-w

[6] Global Internet Metrics with Ookla, https://www.ookla.com

[7] KubeLinter documentation, https://docs.kubelinter.io

[8] Network performance measurements with iPerf, https://iperf.fr/

[9] OpenWrt Project, https://openwrt.org/

[10] Snort Intrusion Prevention/Detection System (IPS/IDS), https://www.snort.org/