Security of Docker containers

Report for the Computer Security exam at the Politecnico di Torino

Carmine D'Amico (239540)
tutor: Antonio Lioy

September 2018

Contents

1 Introduction

Docker is one of the most disruptive technology of the last years. It has almost "revolutionised" the software industry and the method of developing new applications. As stated on the company's site [1] there are more than twenty-nine billion downloads of Docker containers and the project itself has received more than thirty-two thousands stars on GitHub.

Docker promotes a "LEGO approach" for the development of new software applications, in which the Docker images play the role of the basic bricks on which Docker containers are instantiated and started. The contemporary approach to software development is often based on the use of third-party code, libraries or platforms. Docker has led this concept to the extreme, allowing developers to quickly get up and running countless third-party applications. It allows an easier and faster development, facilitating the reuse and sharing of already developed software. Gradually it has taken over other virtualisation technologies like the ones based on the use of hypervisors, thanks to its being lighter and easier to use.

The goal of this research is to analyse Docker from the point of view of the computer security. It starts from other studies on the same topic to explore what security threats are hiding behind Docker's ease and speed of use. Moreover, it derives from such menaces a list of security best practice to follow during the use of Docker containers, analysing different tools and solutions proposed by Docker itself and other companies.

1.1 Limitations

This research focuses solely on Docker. It does not provide a security analysis of other tools that are often used along with it, like Kubernetes [2]. The latter is an open-source orchestration tool, which is used to handle more easily the deployment, the scaling and the management of containers, including also Docker containers.

Moreover, this work wants to propose a security analysis on the current status of Docker. Such software application is constantly developing and new versions are released very frequently. For this reason many scientific paper have not been used, because they refer to obsolete versions of Docker.

1.2 Structure Of The Work

The study is divided as follows:

- Section 2 starts with an excursus concerning the most important technologies used to virtualise computational resources, from hypervisors to containers. Then it focuses on Docker, introducing it and describing its architecture and the most important components involved in its use.
- Section 3 defines the attack surface of the Docker containers. It analyses different security threats, providing for each of them a description and an example of a real attack.
- Section 4 starts from the previous section to derive best practices for the deployment of Docker containers. It analyses different configurations and tools able to mitigate the security threats previously described.
- Section 5 takes in analysis two of the tools introduced previously as best practice to follow. It shows a case of use for each of them, providing a practical example of how to use them from their installation to their evaluation.

2 Docker Overview

In computer science, the term *virtualisation* [3] is referred to the creation of virtual computational resources. These resources, normally supplied as hardware, are instead provided to the user by the operating system through the creation of a new abstraction layer. Operating systems, storage devices and network resources could all be virtualised. Virtualisation can be obtained at different levels and using different techniques.

Virtual machines have represented for many years the state of the art of virtualisation, being used in both consumer and enterprise contexts. In the last years a new technology, based on *containers*, has started to gain more attention specially thanks to its advantages in terms of acceleration of the development cycle and possibility to thicken applications on servers. *Docker* is an open-source container technology which became popular thanks to its ease of use, that allows to create and manage containers in an easy way.

In this section I give an overview of the technologies mentioned above, with a particular focus on Docker, which is the main topic of this work.

2.1 From Virtual Machines...

With the term virtual machines it is often intended an *hypervisor-based virtualisation*, that is a type of virtualisation which acts at hardware level. Virtual machines (VMs) are established on top of the host operating system, providing applications with their dependencies, but also an entire guest OS and a separate kernel. One or more virtual machines can be run on the same machine. Hypervisors are distinguished in two different types, the one that works directly on top of the host's hardware (*bare metal hypervisor*) and the one that is on top of the host's OS (*hosted hypervisor*) (Fig. 1).

Bare metal hypervisor provides better performances, not having the overhead of the extra layer of the host's operating system. It manages directly hardware and the guest's operating system. On the contrary hosted hypervisor can be manged in an easier way running as a normal computer program on the user's operating system [4, Sec. 2].

As said before, the hypervisor needs to run on the user's computer which is defined as *host machine*, while each virtual machine is called *guest machine*. It is important to remember this terminology because it will be used also in the following, referring to containers.

2.2 ...To Containers

Container-based virtualisation represents another approach to virtualisation, mainly spread in the last years. Compared to hypervisor-based virtualisation it is lighter, using the host's kernel to run multiple virtual environments. It virtualises at operating system level (it is also known as *OS-level virtualisation*) allowing other applications to run without installing their own kernel on the host. Containers act as isolated processes just sharing the host's kernel (Fig. 2).

Resources are provided by the host's OS together with the *container engine*. The container engine is the technology in charge of create and manage containers. *Docker* represents one of the most important and most used container engine. A computer program running inside a container can only see the resources allocated to that particular container. On the same host there could be more than one container, each one with its personal set of dedicated resources. Although it could be possible to run more than one computer program inside the same container, it is always suggested to run only one program per container, in order to separate areas of

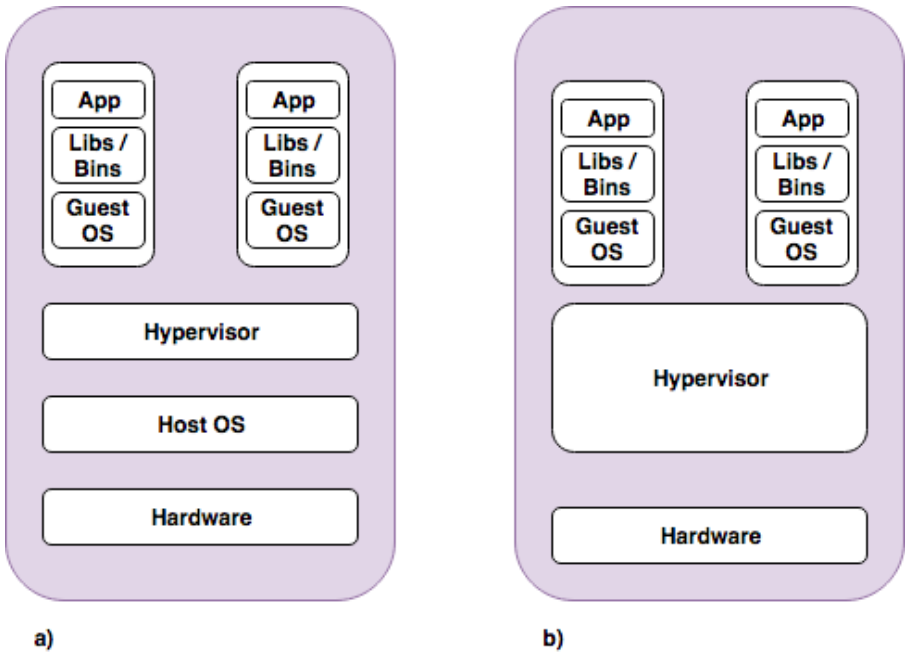

Figure 1: Architectural differences between (a) hosted hypervisor and (b) bare metal hypervisor

concern. It is better to connect multiple containers using user-defined networks and shared volumes. Containers are particularly appreciated inside multitenant environments for their lightness and for their approach to host's resources sharing, which increases average hardware use.

There are many examples of containerisation implementations, like *Linux-VServer*, *OpenVZ* and *LinuX Container (LXC)*. This last implementation will be better described in the next paragraph allowing to better understand the behaviour of containers.

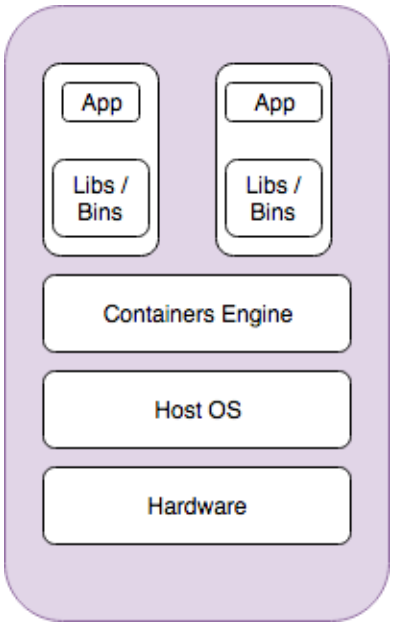

Figure 2: Architecture of a container-based virtualisation

2.3 LXC

LinuX Containers, better known as LXC, are an OS-level virtualisation technique created in 2008. It allows to run multiple isolated Linux instances (the *containers*), on top of a single *LXC host*, which provides a shared Linux kernel [5]. Each container "sees" its own CPU, memory, network interface, I/O, ...

The isolation among containers is obtained thanks to some Linux kernel's tools: *namespace* and *cgroups*. In the following subsections I will analyse these two components and also *Linux kernel capabilities*, both for their importance for containerisation and for the fact that they are also the basic components in Docker.

2.3.1 Kernel Namespace

Namespaces allow to create isolated environments, in which each process that belongs to that particular environment can see global host's resources as personal isolated resources. In other words they allow to create pool of processes that think to be the only ones of the system. In this way groups of processes, that are part of different namespaces, can see different set of resources. Namespaces work by assigning to different resources the same name in different namespaces. In the Linux kernel five different type of environments are implemented [6]:

- **Mount namespaces** isolate the set of file system mount points seen by a group of processes so that processes in different mount namespaces can have different views of the file system hierarchy. With mount namespaces, the mount() and umount() system calls cease to operate on a global set of mount points (visible to all processes) and instead perform operations that affect just the mount namespace associated with the container process.

- **UTS namespaces** isolate two system identifiers (nodename and domainname) returned by the uname() system call. This allows each container to have its own hostname and NIS domain name which is useful for initialisation and configuration scripts based on these names.

- **IPC namespaces** isolate certain inter-process communication (IPC) resources, such as System V IPC objects and POSIX message queues. This means that two containers can create shared memory segments and semaphores with the same name, but are not able to interact with other containers memory segments or shared memory.

- **Network namespaces** provide isolation of network controllers, system resources associated with networking, firewall and routing tables. This allows container to use separate virtual network stack, loop-back device and process space. In this way it is possible to add virtual or real devices to the container assigning them their own IP Addresses and even full iptables rules.

- **PID namespaces** allow processes in different containers to have the same PID, so each container can have its own init (PID1) process that manages various system initialisation tasks as well as containers life cycle. Also, each container has its unique *proc* directory. From within a container only processes running inside the container can be monitored. The container is only aware of its native processes and can not "see" the processes running in different parts of the system. On the other hand, the host operating system is aware of processes running inside of the container, but assigns them different PID numbers.

2.3.2 Cgroups

Cgroups are a kernel tool used to manage resources that belong to different processes. They gather, track and limit the use of resources by processes. It is possible to create and manage *cgroups* using high level code, assigning a PID to a specific *cgroup*. They represent the fundamental tool to obtain resource isolation, playing an important role also for the CPU and I/O's scheduling. The resources that can be limited by Cgroups are [7]:

- **memory** - this subsystem sets limits on memory use by tasks in a cgroup and generates automatic reports on memory resources used by those tasks.
- **CPU** - this subsystem uses the scheduler to provide cgroup tasks access to the CPU.
- **CPUacct** - this subsystem generates automatic reports on CPU resources used by tasks in a cgroup.
- **CPUset** - this subsystem assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
- **blkio** - this subsystem sets limits on input/output access to and from block devices such as physical drives (disk, solid state, or USB).
- **net_cls** - this subsystem tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup task.
- **net_prio** - this subsystem provides a way to dynamically set the priority of network traffic per network interface.
- **ns** - the namespace subsystem.
- **devices** - this subsystem allows or denies access to devices by tasks in a cgroup.
- **freezer** - this subsystem suspends or resumes tasks in a cgroup.
- **perf_event** - this subsystem identifies cgroup membership of tasks and can be used for performance analysis.

2.3.3 Linux Capabilities

Linux kernel capabilities [8] allow to have a fine-grained access control system compared to the simple dichotomy "root/non-root", restricting container's privileges. Using capabilities the concept of root changes its meaning inside a container: a user can be root inside a container, still not having all the normal privileges of a real root user, because only a reduced set of capabilities is allowed. This approach prevents an attacker who manages to escalate as root within a container to have full control of the same. There are different types of capabilities that can be granted or dropped within a container. For example, there are capabilities for managing all the "mount" operations, for accessing the raw socket and for handling operations on the file system.

2.4 Docker

As today, *Docker* represents the most used computer program for operating-system-level virtualisation (containerisation). It is developed by *Docker, Inc* [9] and it was introduced during the 2013's PyCon conference. During its presentation *Docker* was announced as the future of Linux Containers [10], indeed from its first releases it reiterated many concepts from them, such as *namespaces*, *cgroups* and *capabilities*, but providing a simpler user experience and a complete ecosystem to create and manage containers.

Docker's success is mainly addressable to its portability and lightweight nature which allow to create high density environments. It is the ideal software in scenarios where continuous integration and continuous delivery (CI/CD) are required, allowing developers not only to build their code, but also to test their code in any environment type and as often as possible to catch bugs early in the applications development life cycle [11].

2.4.1 History

Docker was born as an inside project within *dotCloud*, a platform-as-a-service (PaaS) company, later renamed to *Docker, Inc*. Solomon Hyckes [12] was the leader of the project, that was at first developed with other *dotCloud*'s engineers, like Andrea Luzzardi and Francois-Xavier Bourlet. The project went public, as said before, during 2013's PyCon conference and it was released as open-source software during the same year. Always during 2013, *Docker* distanced itself from Linux Containers, replacing them with a new execution environment (starting from version 0.9), *libcontainer*.

Docker represented a turning point in the IT industry, as it can be proved by looking ad its adoption. The following is a list from Wikipedia [13] of the milestones achieved by Docker:

- On September 19, 2013, Red Hat and Docker announced a collaboration around Fedora, Red Hat Enterprise Linux, and OpenShift.
- In November 2014 Docker container services were announced for the Amazon Elastic Compute Cloud (EC2).
- On November 10, 2014, Docker announced a partnership with Stratoscale.
- On December 4, 2014, IBM announced a strategic partnership with Docker that enables Docker to integrate more closely with the IBM Cloud.
- On June 22, 2015, Docker and several other companies announced that they are working on a new vendor and operating-system-independent standard for software containers.
- As of October 24, 2015, the project had over 25,600 GitHub stars (making it the 20th most-starred GitHub project), over 6,800 forks, and nearly 1,100 contributors.
- A May 2016 analysis showed the following organisations as main contributors to Docker: The Docker team, Cisco, Google, Huawei, IBM, Microsoft, and Red Hat.
- On October 4, 2016, Solomon Hykes announced InfraKit as a new self-healing container infrastructure effort for Docker container environments.
- A January 2017 analysis of LinkedIn profile mentions showed Docker presence grew by 160% in 2016.[30] The software has been downloaded more than 13 billion times as of 2017.

2.4.2 Docker's Architecture

Docker follows a client-server architecture where three main components can be distinguished [14] (Fig. 3):

- The server, which is a daemon process (called *dockerd*) running on the host's machine. It is in charge of creating and managing Docker's objects.
- A set of interfaces compliant to REST architectural style, that enable programs to communicate with the server, sending instructions.
- A command line interface (CLI) client, that allows the user to interact with Docker using the REST API through their terminal.

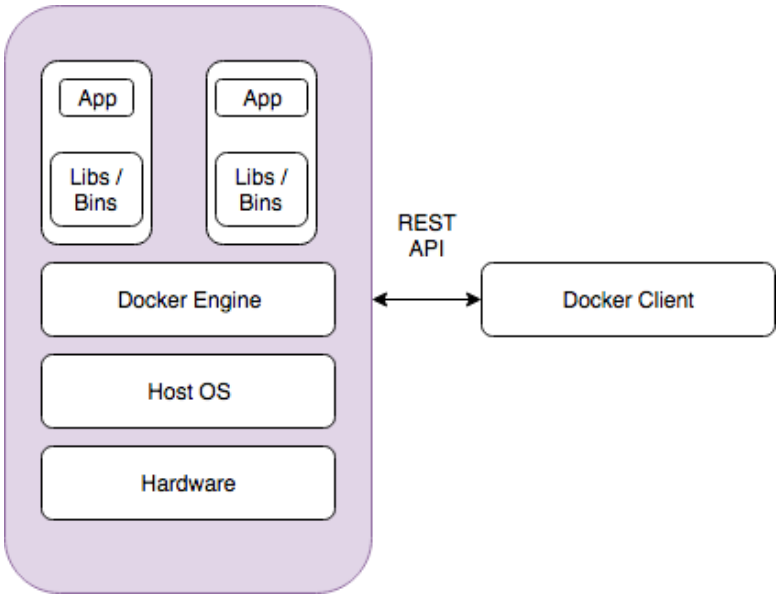

Figure 3: Docker's architecture

2.4.3 Docker's Components

Docker's workflow requires the interaction with many special purpose components created and managed by *dockerd*:

- **IMAGES** are the basic components involved in the creation of a Docker's container, they include all the instructions that the daemon has to follow in order to run a container. An image can be created from scratch or it can be based on already existing images (for example on the image of *nginx* if we want to create a web server) where the other needed components are installed.

  A *Dockerfile* is a special file that follows a very simple syntax, which includes all the steps that must be followed in order to create an image. Each instruction represents a layer in the image. When we insert a new layer (modifying the *Dockerfile*) rebuilding a container, only the new layer is rebuilt, speeding up the deployment's process. An image is built from a Dockerfile using the `docker build` command.

  Another way to create an image is to run an already existing container, perform all the modifications needed and at the end save the status achieved as a new image with the `docker commit` command.

Docker images are stored inside registries which can be private or public. The two most famous public registries are *Docker Cloud* and *Docker Hub*: the latter is the default one visited by Docker for searching images. Developers can build their own images and upload them to the Docker Hub, or they can just download already existing images from it. Developers' images on Docker Hub are by default public and only the paid accounts can upload private images. These images take a standard name, that has the form "developer_uid/repository_name". *Docker, Inc.* also provides some official images.

A Docker image can also contain a *tag*, that is just an ID that conveys information about a specific image version/variant. A tag can be added to an image using the command `docker tag`. If no tag is specified, the *latest* tag is assigned by default.

- **CONTAINERS** represent the running instances of images. The relationship between images and containers could be compared to the relationship between classes and objects in an object-oriented programming language like Java. A container's behaviour can be set up starting from its image or by the configurations specified at launch time: indeed, a container can be launched using the `docker run` command.

  When a container is started Docker searches locally for all the needed images (downloading them from online public registries if necessary), then a read/write file system is allocated, where the container can create or modifies file or directories. By default a container can be connected to the external network using the host's connection. When a container is stopped any changes to its state, that are not stored in persistent storage, disappear.

- **SERVICES** are supported from version 1.12 of Docker. Like in a distributed system, services represent the different pieces of an application. In particular in the context of Docker a service is just a running container in which it is defined the way that its image will be executed. Through services indeed a user could configure the port which a container will use, the number of replicas that will run of such container to better scale an application, and so on.

  Services can be configured using a special Docker file: `docker-compose.yml`. This file contains all the instructions that must be followed to run the application's services. For each service indicated it specifies where to pull the corresponding Docker image, the port mapping between the container and the host, the load-balanced overlay network that will be used and all the information needed to deploy such service (the number of replicas, the amount of resources needed, the restart policy, etc.).

- **SWARMS** are cluster composed by machines that are running Docker. A *swarm manager* is a machine belonging to the cluster and in charge of executing the commands received. A user continues to run Docker commands normally as described before, the difference is that such commands are executed by the swarm manager, which decides how to run these commands (for example filling the host with less running containers or assigning to each host one instance of the specified container). In a cluster there could be more than one swarm manager. *Workers* are machines belonging to the swarm, but without the same privileges of a swarm managers. They just provide more computational capacity to the cluster. In general all the machines inside a Docker swarm are referred as *nodes*.

2.4.4 Common Usage Of Docker

As stated in the previous paragraphs, Docker's success is mainly attributable to its simplicity of use and to its "lightness". Such characteristics have increased its adoption among developers

and organisations. John Willis, a Technical Evangelist at Docker, during an interview [16] for the IBM Infrastructure Blog stated that the most common use cases for Docker are mainly three:

- *Integration test*: virtual machines running integration tests could take also days to complete the job. The virtual machines should be configured and installed, then after the tests they should also be rebased back to their original state. On the contrary with Docker the configuration and the execution of a container takes less time and also rebase could be done in few seconds.
- *Immutable delivery model*: with Docker when a developer needs to commit its code he can just commit a Docker image. In such a way the software tested on his own computer will be the same identical software which will run in production. This principle allows to speed up the deployment process, not requiring further configurations to run the software on the production hosts.
- *Container as service*: Docker allows to build up a collection of pre-built binaries (Docker images) that can be shared, for example pushing them into the Docker Hub. In such a way an organisation can speed up its production and also increase its set of used tools, not needing to set up its own tools but just to pull new images.

3 Docker's Security Threats

In this section I analyse the most important security threats for a Docker system. For each possible threat I describe its characteristics and how it could be possible to take advantage of it. I also provide a practical example of an attack which exploits the vulnerability described, exception made for those threats that are more generic, not regarding specifically Docker containers.

In the next section I will start from these security threats in order to define the best practices to follow using Docker.

3.1 Analysis Of Docker Security Threats

In order to list the main security threats that afflict a Docker system, I started from a possible definition of the attack surface for the Docker containers. I have analysed a series of studies present in literature, from Yasrab's paper "Mitigation Docker Security Issue" [17] to "To Docker or Not to Docker: A Security Perspective" written by Combe, Martin and Di Pietro [39], through Sysdig's list of the well known vulnerabilities of Docker [18]. Finally, I have analysed the Docker attack surface from five macro view points:

- *Isolation*: the most important difference between a normal process running on a host and a Docker container is the degree of isolation of the latter. As seen in the previous section, Docker uses different Linux kernel features to achieve an optimal isolation. However, such tools must be well configured, because it is possible that their default behaviour leads to *denial-of-service attacks*, *container breakouts* or *ARP spoofing attacks*.
- *Host Hardening*: Docker containers share the kernel with the host making them lighter than a virtual machine. Such characteristic increases the importance of the host's kernel security, cause both the containers and the host depend on it. *Kernel exploits*, for example, can lead to malfunctions of both the host and the containers.
- *Image Security*: Docker images are fundamental for the correct execution of Docker containers. If a *poisoned image* is downloaded and used, an attacker can compromise the entire system. In the same way the use of *outdated Docker images*, can lead to bugs in the system easily exploitable by attackers. For this reason, maximum precaution must be taken managing Docker images.
- *Erroneous Configurations*: Docker has made of its ease of use its most important feature. However some precautions must be taken configuring a Docker system. For example an *erroneous management of the secrets* can bring to a tampering of the system.
- *Running Container*: inside a Docker container there is anyway a running program. All the bugs and vulnerabilities that afflict such program can be used by an attacker to hit the system. For this reason, to cover all the attack vectors of Docker containers, *monitoring the runtime activities of containers* is mandatory.

From these macro view points, I have extracted eight security threats for Docker containers, analysing them in the following paragraphs.

3.1.1 Kernel Exploits

One of the main advantages of container-based virtualisation, so of Docker, is that the host shares its kernel with all the running containers. In this way this kind of virtualisation results lighter than an hypervisor-based one, avoiding the overhead of installing the guest's kernel. As we will see this is on the one hand a big advantage in terms of speed and efficiency, but on the other hand it represents a big threat for the security of the system. The host's kernel, indeed, handles all the operations of the containers, so in case of a *kernel-level exploit* all the containers that are running on the system are at risk of being compromised.

Kernel exploits can be done by an attacker who takes advantages of a bug or a vulnerability of the kernel. In this way he can run his software in kernel mode, manipulating processes' privileges and bringing him to take control of the system. If such exploit is performed inside a container, it has consequences also on the host OS. In particular if the attack allows the attacker to execute his code, this execution will happen on the host OS, not inside the container; in the same way if the attack allows to read arbitrary memory, the attacker could read and write memory parts that belong to other containers and to the host itself.

Kernel exploits are also a security threat in the context of an hypervisor-based virtualisation, but in that case the attack would result more complicated. Indeed the attacker should be able to exploit the VM's kernel, the hypervisor and the host's kernel. While on a container-based virtualisation it is sufficient to exploit the only host's kernel.

Practical Example

In literature there are many examples of kernel exploits used to obtain control over an entire system. In this paragraph I will focus on *Dirty COW*, an attack that takes advantage of a privilege-related vulnerability in order to allow the attacker to gain high privileges on the system. The official reference to this bug is CVE-2016-5195.

It exploits a race condition inside the Linux kernel's memory subsystem that handles the copy-on-write (COW) mechanism, in this way an unprivileged user could gain write access to a read-only memory part [19].

In order to obtain this exploit, we must create a loop involving two threads:

- one thread tries to access inside a read-only memory location to write, creating in this way a modified copy inside the process's memory;
- another thread, calls the syscall *madvise()* [20] with the MADV_DONTNEED parameter (such parameter indicates that it is not to be expected to access the memory location in question) for the newly allocated memory.

The simultaneous execution of these two threads in a loop could bring the kernel to point to the modified copy of a file in memory that should instead be read-only [21].

3.1.2 Denial-of-service

As we have seen for kernel exploits, the fact that all the containers running on a host share the same kernel can be both positive and negative at the same time. All the containers, indeed, share also the same resources and if the access to them is not limited in some way one container could require a huge amount just for it, bringing the host and the other containers to starvation. This type of attack is known as *denial-of-service(DoS)*.

Denial-of-service attacks are very well-known attacks in literature, especially in multi-tenant systems, that are the ones where Docker is mostly used. The aim of the attack is to make a system or a network resource unavailable. It is typically accomplished by flooding the targeted machine or resource with superfluous requests in an attempt to overload systems and prevent some or all legitimate requests from being fulfilled [22]. An attack of this genre, conducted against a Docker container, brings to unavailability not only the container itself, but also the host where it is virtualised together with all the other system's containers.

On a virtual machine this type of attack is also possible, but it is more difficult to be completed. This is due to the fact that the hypervisor is configured to restrict its use of resources. For a container, instead, resources management is defined at application layer.

In the previous section we have seen how Docker has inherited cgroups from LXC to allocate resources needed by containers. However, as described on Docker's documentation [23], this kernel tool is not enabled by default, so a container can use as much of a given resource as the host's kernel scheduler allows. We will see in the next section how to configure Docker to work with cgroups, managing resources for containers.

3.1.3 Container Breakout

If a user inside a Docker container bypasses all the isolation checks, "escaping" from the container, it would have direct access to the host and to all the other containers in the system. This situation can be achieved using an exploit or a not correct configuration of the Docker environment. Such type of attack is known as *container breakout*.

This type of vulnerability, according to Docker website [24], was mitigated with Docker 1.0; nevertheless a not trusted program running with root privileges inside a Docker container could still represent a source of risk.

Container breakout is mostly possible in situations where privileges inside a container were not properly configured. By default, users are not namespaced, so any process that breaks out of the container will have the same privileges on the host as it did in the container. A root user in a container will also be root on the host. Moreover, in the earliest versions of the Docker engine, some specific kernel capabilities were dropped, but still leaving some important ones granted. From Docker 1.0, instead, a different approach was taken: all the capabilities were dropped, except some necessary, giving the user the possibility to decide which one to allow. Actually this newly approach transformed the use of capabilities from a blacklist to a white-list.

Practical Example

One of the most famous container breakout exploit is 2014's *Shocker* [25]. I will analyse such attack with the aim to demonstrate how a Docker container can access some privileged file system data when Linux capabilities are not managed carefully.

Shocker takes advantage of the *CAP_DAC_READ_SEARCH* capability, that was granted by default to a superuser inside a container with a version of Docker prior to the 1.0. This capability is the one which allows to use the system call *open_by_handle_at()*, conceding to access a file on a mounted file system through a file_handle structure. A file_handle is quite different than a file descriptor, because it can be generated inside a process using *name_to_handle_at()* and then recalled by another process with *open_by_handle_at()* (while a file descriptor can't be properly passed between different processes). In this way if a process has an handle opened for a host's file or a different container's file and our container has not the privileges to access it, we can still call *open_by_handle_at()* on the opened handle [26].

3.1.4 Poisoned Images

Docker images represent one of the most fundamental building blocks for a container. If an attacker obtains to run his malicious code through a tampered image, the host and all its containers can be seriously at risk. For this reason the problem of ensuring the provenance and the integrity of an image is of primary importance in the use of Docker.

The term *poisoned image* is referred to an image that has been injected with malicious code by an attacker. In a typical attack situation a victim will download such poisoned image through the `docker pull` command from a registry. The private registries are those most interested in this attack, because very often they are easily exploited due to their weak security settings. The `docker pull` command plays a key role in such threat, indeed it is used to fetch and unpack a container image in just one step. As explained by Jonathan Rudenberg in a post [28] on his blog, such mechanism and its related pipeline, even if functional, can lead to serious security risks. The Docker images downloaded via the `docker pull` command go directly through a processing pipeline inside the Docker daemon consisting of the following operations:

$$\text{DECOMPRESS} \rightarrow \text{TARSUM} \rightarrow \text{UNPACK}$$

A poisoned image can compromise a system directly during such operations, even if the container itself is never run. Moreover, this pipeline does not contain any verification step for the downloaded image, thus not verifying provenance of the image itself.

Practical Example

As stated, the `docker pull` command follows an unsafe pipeline, which can lead to compromise the system without even running the container relater to the poisoned Docker image. A report by Red Hat [27] describes how many vulnerabilities against Docker exploited such mechanism. For example a method widely used by the attackers, now resolved, was to perform *directory traversal attacks* during the unpack phase of the pipeline previously reported: a tarball's capacity was used to unpack to any possible path like "../../../", in this way a poisoned image could have overridden any part of the host file system.

CVE-2014-9357 [29] was another known vulnerability, solved from Docker 1.3.3, that allowed attackers to execute arbitrary code with root privileges via a malicious image or a build from a compromised Dockerfile. Such vulnerability exploited the fact that *xz* utility, used for decompressing LZMA archives during a call to *docker pull*, was executed as root.

3.1.5 Outdated Images

The importance of images in Docker ecosystem makes it of primary importance to attest their security. On the one hand we have seen how images could be *poisoned* by an attacker, on the other hand even if our images are not compromised it does not mean that they are safe. It is important to make sure that the images at the base of our running containers are updated, not containing any known vulnerabilities. An outdated image can be affected by a series of security threats that were already fixed in an updated version of the same image.

A study by Gummaraju, Desikan, and Turner at BanyanOps [30] demonstrated how about more than the 30% of the official repositories present on Docker Hub are affected by a variety of security threats, such as *heartbleed*, *shellshock*, *poodle*, etc. These numbers grow up to the 40% taking in consideration also the images loaded by users.

3.1.6 Management Of Secrets

Docker containers are very used in the development of microservices. A microservice architecture is very different than a monolithic one, specially in the deployment phase: a monolithic architecture is often just configured, launched and then it runs for a long period of time (which could even last years); microservices instead are continuously created and destroyed. In both cases, sensitive information are needed, like API keys, database passwords, SSL/TLS keys, SSH keys, and so on. Compromising these information would compromise the entire system.

In a monolithic architecture the management of these information is non trivial, indeed they can be stored in the system permanently and they can be renovated using some mechanism like the "Privileged Accounts Managers" [31]. Despite these solutions have been used and tested for years in the context of monolithic service, they can't be applied to microservices based on containers. The two main concerns for the management of secrets inside Docker containers are the following[32]:

- Docker images have an immutable nature, this means that they are created once and then deployed in many different environments. Their nature is strongly in contrast with the idea of saving secrets directly inside them;
- requesting such secrets at runtime imply performing a prior authentication procedure, but this procedure is difficult to be implemented without storing some secrets for the authentications itself.

The use of environment variables is highly discouraged, due to the fact that these variables can be easily leaked. They are indeed exposed in too many places, like child processes, linked containers and *Docker inspect* (a tool provided by Docker for retrieving low-level information on the used objects).

All these reasons make *management of secrets* a central topic in the discussion of Docker's security.

Practical Example

One example of how a bad management of secrets could be fatal for an agency comes from IBM. In 2017 a privilege escalation vulnerability was found inside IBM Data Science Experience, a data analytics product [33]. Such vulnerability was caused by a wrong configuration of the Docker containers that were running the service. IBM's engineers left inside the containers Docker TLS keys, leaving root access across the whole computer cluster and read/write access to terabytes of sensible customer data.

As stated by the report of the vulnerability [34], an exploit of such threat was also quite easy, requiring only an access to to Internet and a web browser:

- the attacker should just enter the service's web environment, accessing to its command line;
- he should download and extract the Docker binary of the service, using the following commands:

```
system("wget
   https://test.docker.com/builds/Linux/x86_64/docker-1.13.1-rc1.tgz")
system("wget
   https://test.docker.com/builds/Linux/x86_64/docker-1.13.1-rc1.tgz")
```

- he should use the downloaded Docker binary with the existing certificates to achieve root access to the host mounted volume:

```
system("DOCKER_API_VERSION=1.22 ./docker/docker -H 172.17.0.1 \
        --tlscacert /certs/ca.pem --tlscert /certs/cert.pem \
        --tlskey /certs/key.pem \
        run -v /:/host debian cat /host//shadow")
```

3.1.7 ARP Spoofing

Inside an host all the containers, if not properly configured, can communicate with each other through their network interfaces. Docker uses namespaces to create an independent network stack for each container, giving them their own IP addresses, IP tables, etc. By default Docker provides connectivity between the containers creating a Virtual Ethernet Bridge, named *docker0*. At container creation time, Docker creates and connects a new network interface with a unique name to the bridge. Such interface is also connected to the *eth0* interface of the container (Fig. 4). This type of configuration is vulnerable to *ARP spoofing* attacks, since the bridge forwards all of its incoming packets without any filtering.

An ARP spoofing attack is a very well known attack in literature. It takes advantage of the ARP protocol, which doesn't provide a basic method to authenticate ARP messages. In order to perform such type of attack, the attacker must be connected directly to the target LAN with his machine or to a compromised host that belongs to the network. In the case of Docker, such conditions is satisfied if the attacker manages to compromise a container running on a host: in this way he would have access to the local network between all the containers running on the host. The main goal of this type of attack is to divert traffic directed to a container to the compromised container controlled by the attacker.

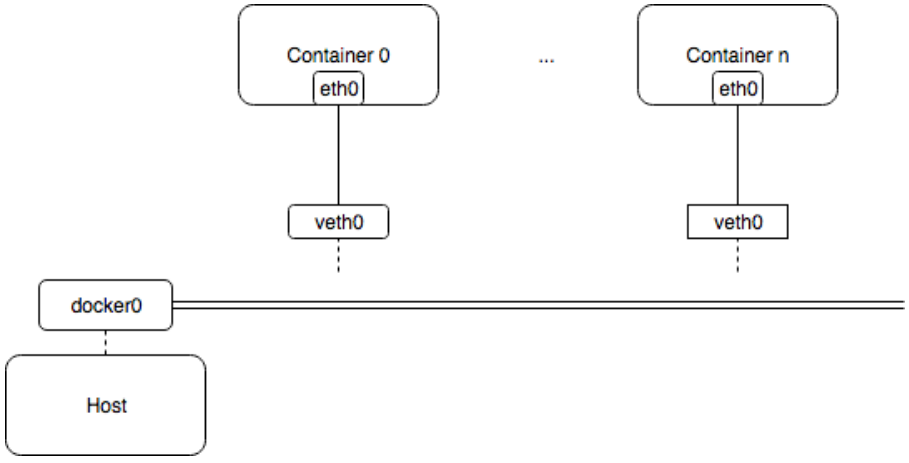

Figure 4: Docker default network model

Practical Example

A practical example of this type of attack can be reproduced inside a containerised environment just running a Docker container with *dSniff* installed. Dsniff is a set of password sniffing and network traffic analysis tools, which contains among others also *arspoof*, a tool used specifically to perform ARP spoofing attack [35].

Such type of experiment is described by Philipp Bogaerts in one of his blog post [36]. He shows how it is simple to perform an ARP spoofing attack running three containers on the same host, without changing the default network configuration:

- two containers are created just from the Busybox base image, running the "ifconfig" command on each in order to read their IP address. We will call these two containers just *container_0* and *container_1*;
- one container, which will be the one that performs the attack, is created starting from the Debian base image, in which it is installed only dSniff.

One of the first two containers ping the other, while the attacker's container performs the attack using arspoof:

```
arpspoof -i eth0 -t ip_container0 ip_container1 &
arpspoof -i eth0 -t ip_container1 ip_container0 &
```

In such a way all the traffic generated between the first two containers will pass through the attacker's container.

3.1.8 Dynamic Aspects of Docker Security

All the threats analysed in the former sections regard the launch time of containers. Kernel exploits, denial-of-service, container breakouts, etc. are all security threats that can be faced before running our containers. We can refer to these as *static aspects* of Docker security. Such threats, as we will see in the next section, can all be mitigated following some precautions during the creation of the images that we will run, using special configurations and ad-hoc policies or tools.

However container's attack surface is not limited to such aspects, as we must also consider:

- all the possible vulnerabilities of the application that we want to run inside a container, that can be exploited by an attacker;
- all the attacks that can be performed on a vulnerability already discovered but not yet patched (these attacks are also known as zero-day attacks).

Such attacks can not be avoided at launch time, but they must be mitigated at run time, monitoring containers' activity . We can refer to them as *dynamic aspects* of Docker Security.

4 Best Practices For Docker Deployment

As seen in the previous section, the ease of use of Docker actually hides a series of security threats. In this section I will start right from these security threats previously analysed in order to list several best practices that can be followed during the deployment of Docker containers. The aim of using such practices is to increase the security of a Docker system, trying not to complicate the Docker user experience. Most of the practices that I will introduce refer directly to specific attacks previously described, others are generic practices that can be useful for increasing the security of the system.

The suggested tools are almost always either provided by Docker itself or open-source and free; I deliberately avoided to propose commercial and closed-source software because it is often more difficult to analyse and test. Most of the practices are present in literature or directly suggested in the Docker's documentation. In particular I have often referred to Gianluca Arbezzano's "Docker Security - Play Safe" [40] e-book and to Adrian Mouat's "Using Docker. Developing and Deploying Software with Containers" [41] paper to find the best practices for security enhancement in Docker.

4.1 Mandatory Access Control

Mandatory access control systems are well-known in the literature and often their use is recommended in conjunction with Docker. Their use is proposed in many studies from authors such as *Yasrab* [17, Sec. 4.1], *Combe et al.* [39] and *Raj MP et al.* [37, Sec. 3.a-b].

A mandatory access control or *MAC* system is a set of rules that define what a subject can or cannot do in reference to a certain object. Usually the operating system plays the role of the policy administrator; threads and processes are the subjects; files, directories, network ports, and so on are the objects of the defined policies. When a subject tries to perform an action onto an object the OS checks all the defined policies and decides if the action can be done or not. A user cannot override or modify these policies, only the policy administrator can [42]. Linux uses by default *discretionary access control (DAC)*, restricting access to objects based on the identity of subjects and/or groups to which they belong (authorisations are indicated on each file/directory by the `rwx` flags). MAC and DAC can works together on a Linux machine, with DAC that must be passed before MAC evaluation.

Linux offers some kernel security modules, like *Security-Enhanced Linux (SELinux)*, *AppArmor* and *Secure Computing Mode (seccomp)*, that can be configured using access control security policies to implement mandatory access control. Docker supports by default these modules and by using them we can have an additional extra layer of security, which can result decisive to mitigate kernel exploits and container exploits. An example of this can be found in Jon-Anders Kabbe's work "Security analysis of Docker containers in a production environment" [38, Sec. 4.2.1], where he demonstrates how Dirty COW, the kernel exploit analysed in the previous section, can simply be prevented using Docker's default profile of AppArmor.

The general policy of Docker's default profiles for the described Linux kernel security modules is usually to protect the host from the containers and not also containers from other containers [39]. However writing specific profiles for each container running on the system can address such problems, making MAC very useful for the enhancement of containers isolation and for the prevention of kernel exploits and container breakouts.

4.1.1 SELinux

Security-Enhanced Linux, better known as SELinux, is a Linux kernel security module created in 2003 and used to develop a mandatory access control system. With SELinux each component of our operating system has a *label* (SELinux is often known as a labelling system): process, files, directories, devices, network port, etc. are all labelled. To achieve mandatory access control it is necessary to write rules that control the access of a subject label onto an object label. By default everything is denied and by writing policies some specific actions can be allowed.

The use of SELinux with Docker allows to create access policies that can increase containers isolation. To run Docker with SELinux we must at first install such kernel module on our host, then we have to assign labels accordingly to the policy that we want to create. Docker has to be launched with the `--selinux-enabled` flag to start in order to use SELinux.

SELinux is a very powerful tool that allows to have a great control over the system, through its high granularity access policies and all its many configurations. At the same time its fine-grained policy maker system makes it very difficult and complex to maintain. For this reason other MAC tools are often preferred.

4.1.2 AppArmor

AppArmor is another Linux kernel security module based on mandatory access control like SELinux. It offers the possibility to restrict programs' capabilities with per-program profiles. The system administrator can create and load a security profile into a single program, this makes AppArmor simpler to use than SELinux and this is often the reason why it is preferred over it. AppArmor can be used in two different modes:

- *Enforcement mode*, where the defined policies are followed and the access restricted according to them;
- *Complain/learning mode*, where the defined policies are not followed, but each violation is logged for debugging.

We can load our pre-configured AppArmor profile with Docker at launch time, if our host supports such kernel security module. The profile is loaded directly into the container in enforcement mode. If no profile is specified Docker uses its default one. AppArmor profiles are simple text files where rules for capabilities, networking and access to certain objects can be specified. Such rules can be referred to various access controls, denying or allowing a subject to perform certain actions on an object. Security profiles can be saved in /apparmor.d/ and they can be loaded into a container using the flag `-security-opt=‘‘apparmor:profile_name’’` with the `docker run` command.

Bane

Bane [43] is an open-source tool created by Jessie Frazelle, a former Software Engineer at Docker Inc., with the aim to simplify and speedup the creation of AppArmor profiles. It basically generates AppArmor profiles from YAML specification files. Such YAML files follow a very simple syntax where three sections are present:

- *File system*, where we can define the type of actions that can be performed on a certain path. Actions include "Read only", "Log on write", "Allow execution", "Deny execution", "Write", and so on. An example of section of this type is the following, where `/bin`, `/dev` and `/mnt` are accessible only for reading, the `/home` folder can be accessed for writing, the only process that can be executed is *nginx* and the execution of `/bin/sh` is denied:

```
[Filesystem]
ReadOnlyPaths = [
  "/bin/**",
  "/dev/**",
  "/mnt/**"
]

WritablePaths = [
  "/home/**"
]

AllowExec = [
  "/usr/sbin/nginx"
]

DenyExec = [
  "/bin/sh"
]
```

- *Capabilities*, where a whitelist of the allowed capabilities is defined. An example of section of this type is the one that follows, where the only capabilities allowed are *chown* and *net_bind_service*:

```
[Capabilities]
Allow = [
  "chown",
  "net_bind_service"
]
```

- *Network*, where allowed protocols and network policies are defined. An example of network section is the following, where the only protocols allowed are TCP, UDP and ICMP:

```
[Network]
Protocols = [
  "tcp",
  "udp",
  "icmp"
]
```

Bane directly installs the generated profiles inside AppArmor's directory.

4.1.3 seccomp

Secure Computing Mode, better known as *seccomp*, is a Linux kernel security module that acts as a firewall for system calls. Writing a profile for seccomp it can be possible to restrict the use

of syscalls using a whitelist approach. Docker's default profile for this feature disables around forty system calls, not allowing a container to call them, restricting its privileges. It is possible to write specific security profiles for the containers, however such practice is discouraged due to the difficulty to write and maintain seccomp's profiles. seccomp's profiles can be used within a container using the flag `-security-opt=‘‘path_to_seccomp_profile’’` with the `docker run` command.

4.2 Host Hardening

As well as for mandatory access control tools described in the previous paragraph, there are many other security systems that can be used to harden a Docker host. Such systems don't require Docker-specific configurations and can work without interfering with other Linux tools used by default by Docker, like capabilities.

In this paragraph I analyse *grsecurity*, one of the most famous and known set of patches for the Linux kernel which emphasises security enhancements.

4.2.1 grsecurity

grsecurity [44] is a collection of security features for the Linux kernel. It is developed by open-source Security and it can be used only by its paid subscribers, despite being free from 2001 (when it was first released) to 2005.

grsecurity offers different components that add safety checks, useful to defeat many exploits. Among these components the most important are:

- *PaX* [45]: It is used for implementing least privilege protections for memory pages and for reducing the risk of memory corruption bugs. PaX provides hardening with different tools:
  - *executable space protections*: it prevents those attacks where malicious code is inserted into the address space of a process and then launched. Such attacks exploit the fact that Linux by default allows programs to change their memory protection. On the contrary PaX denies memory mappings to be altered from their initial state, in this way a memory portion can not be executed after being written by an user;
  - *address space layout randomisation (ASLR)*: it is used to randomise the memory map of a process. In this way every time that a process is launched its memory map is different. Such mechanism prevents an attacker from finding its malicious code within an address space;
  - *miscellaneous memory protections*: PaX has different features used to protect memory, like: erasing the stack before returning from a system call; preventing overflows from various object reference counters; enforcing the size of heap objects when they are copied between the kernel and the user's address space; etc.
- *Role-Based Access Control (RBAC)*: it is a set of rules to achieve access control through the definition of a series of roles. Each subject of the system has a proper role among with its restrictions on what it can do or not do. The aim of role-based access control is to have roles with the absolute minimum privileges to work correctly and nothing more. In this way it is more difficult for an attacker to take control of the system and to access sensitive data;

- *Chroot restrictions*: they reduce the possibility of privilege escalation attacks arising from the use of the system call *chroot*. Such restrictions include several prohibitions that deny the possibility to attach shared memory outside chroot, to send signals by fcntl outside chroot, to view any process outside chroot, etc.;
- *Audit tools*: they allow to have a complete log of specific group of users. In particular they can be used to log calls to *chdir*, mounting/unmounting of devices, changes to the system time and date, failed *fork* attempts, etc.;
- *Trusted path execution*: it restricts the use of binaries not owned by the root of the system, in this way an attacker can not execute his malicious code.

4.3 Management Of Secrets

As described in the previous section, we can not manage secrets in Docker as we do with the classic monolithic web service. They can not be stored in a Dockerfile or in our application's source code. In this paragraph we will see a series of tools that can be used to achieve a correct management of the secrets. The first, *Docker secrets*, is proposed by Docker in its documentation [46], while the second, *Vault*, is suggested by the *Katacoda* [47] site in one of its courses about Docker.

4.3.1 Docker Secrets

Docker Secrets was released with version 1.13 of Docker. It can be used with Docker swarms to create a central built-in security database, where secrets can be stored and transmitted to only those containers that need access to them. Secrets are encrypted both during their transmission and storage inside a Docker swarm.

When a secret is added to a Docker swarm a bidirectional TLS connection is created between it and the swarm manager. The secret is at first encrypted and stored inside the swarm manager, then it is replicated among all the the other managers of the cluster to ensure its availability. A node of the cluster can require a secret only if it is a manager of the swarm or a running service task which have been granted access to the secret. When the secret is transmitted to a node that requires it, it is decrypted and stored inside the container in an in-memory file system. When the node stops running its stored secrets are unmounted and flushed from the container's memory.

A secret can be added to Docker using the command `docker secret create secret_name *secret*` and the it can be included inside a container using the code `--secret secret_name` with the `docker run` command.

4.3.2 Vault

Vault [48] is a tool developed by HashiCorp for securing access to secrets. It can save secrets to disk or to other persistent services, including also Hashicorp's own backend system *Consul* [49].

Vault offers different features that can be useful in order to achieve a correct and secure management of secrets:

- secrets are encrypted before being written to a persistent storage. In this way an attacker that achieves to access to the storage cannot obtain secrets anyway;

- each secret has an associated lease. An user can renew such lease using Vault's API. At the end of the lease Vault process to revoke the associated secret;
- secrets can be generated and revoked dynamically for some of the most used web services, like AWS or SQL databases. When an application needs to access to one of these services Vault generates a keypair with valid permissions on demand. At the end of the lease the keypair will be revoked automatically from Vault itself;
- in case of key rolling or of a detected intrusion in the system, Vault is able to revoke entire trees of secrets. For example all the secrets accessed by a user or that belong to a specific category.

As stated Vault can use Consul as backend: both such services can be used by a process inside a Docker container for managing secrets and they can also be deployed as Docker containers themselves. Vault can be initialised from its container using the `init` command, which creates a file with the information to access the vault. Such information contain five master keys and an access token that should be stored separately and securely offline or using a third-party service. Vault starts at first in a *sealed* state, in which it can communicate with the backend but it can not decode the contents retrieved. To decode the data we must go through a *unsealing* process. During such process three of the five master keys previously generated must be provided to the Vault process, using the `unseal` command. Finally to access to Vault for reading and writing data the access token must be used to authenticate.

4.4 Resource Limitation

Denial-of-service attacks can be a serious threat for Docker containers as we have seen in the previous section. In order to prevent such attacks we must make sure that the use of resources by containers is limited. Docker suggests its own ways [23], based on the use of cgroup, to manage how much memory, CPU or block I/O a container can use.

4.4.1 Memory Limitation

Docker offers two possibilities for limiting the use of memory: *hard limit* and *soft limit*. The former allows to indicate the maximum amount of memory that a container can use; the latter allows a container to use as much memory as it needs until Docker detects contention or low memory on the host machine.

Memory limitation can be set using runtime configuration flags of the `docker run` command:

- `-m` or `--memory=` indicates an hard limit to the amount of memory that a container can use;
- `--memory-swap` can be used if an hard limited has been set. It indicates how much memory a container is allowed to swap to disk. If a positive number is indicated, it represents the total amount of memory and swap that can be used. If the indicated number is equal to the one specified as hard limit, the container can not access to swap. By default (if this flag is unset) the container can access twice as much swap as the hard limit indicated;
- `--memory-reservation` indicates a soft limit to the amount of memory that a container can use. If this flag is used in combination with an hard limit, the amount of memory specified must be lower than the one indicated with the `--memory` flag.

An integer number must follow such flags to indicate the amount of memory, together with one of the following suffix: `b` (byte), `k` (kB), `m` (MB), `g` (GB). For example the following command can be used to run a container with the memory limited both by an hard limit of 2 GB and a soft limit of 1 GB:

```
docker run --memory=2g --memory-reservation=1g image_name
```

4.4.2 CPU Limitation

Docker allows to set different constraints in order to limit a container's access to the host's CPU. As for memory also CPU limitation can be set using runtime configuration flags of the `docker run` command:

- `--cpus=< value >` indicates the amount of CPU resources that a container can use. The value indicated is proportional to the number of CPUs that a host has;
- `--cpuset-cpus` indicates the specific CPUs or cores that a container can use. An hyphen specifies a range (for example 0-2 indicates to use from the first to the third CPU/core), while a comma indicates a list (0,2 means to use the first and the third CPU/core);
- `--cpu-shares` assigns a *weight* to a container when CPU cycles are constrained. A weight is used to prioritise container CPU resources for the available CPU cycles. The default weight for a single container is 1024.

For example the following commands can be used to run two containers, that can simply be called *A* and *B*, which have 70% and 30% respectively of the available CPU resources:

```
docker run --cpu-shares 716 image_name_A
docker run --cpu-shares 307 image_name_B
```

Where 716 is the 70% of 1024, while 307 is the 30%.

4.4.3 I/O Limitation

As seen for memory and CPU, Docker can limit also I/O for a container. The runtime configuration flags that can be used are:

- `--device-read-bps` indicates the limit (in bytes per second) with which a container can read from a device;
- `--device-read-iops` indicates the limit (in I/O per second) with which a container can read from a device;
- `--device-write-bps` indicates the limit (in bytes per second) with which a container can write to a device;
- `--device-write-iops` indicates the limit (in I/O per second) with which a container can read to a device.

For example the following command can be used to limit the write speed of a container to `/dev/sda` to 50 MB/s.

```
docker run --device-write-bps /dev/sda:50mb image_name
```

4.5 User Namespace

Most of the privilege-escalation attacks can be prevented running Docker containers' application as unprivileged user. At the same time, however, some applications require explicitly to run as root within their containers. In these cases it can be useful to *remap the root user* of a Docker container to a less privileged user on the host, as suggested by the Docker documentation [50]. Such technique is possible thanks to the use of Linux namespaces.

The namespace remapping is achieved through two important files: `/etc/subuid` and */etc/-subgid*. These files contain an entry for each user in the system and each entry is composed by three fields that represent: the ID of a user, his beginning UID (in /etc/subuid) or GID (in /etc/subgid) and a maximum number of UIDs or GIDs available to the user. *dockermap* is the default ID used by Docker to represent the unprivileged system user that is used for the remapping. We can manually create this entry inside the two files, specifying an arbitrary range for the UID/GID (paying attention to not overlap the range of the new user with the already existing ranges of the other host's users). At this point we can configure the Docker daemon to run containers using the just created user. To do this we can edit Docker's JSON configuration file `/etc/docker/daemon.json` appending the option: `''userns-remap'': ''default''`.

It is important to pay attention to the fact that if there is any location on the host where a container can write, permissions must be adjusted accordingly for the dockermap user.

4.6 Image Creation

Creating Docker images is an important aspect in the use of Docker. Follow some basic rules during this process can be fundamental for the system in terms of cost of maintenance and security. We can divide such process into two key steps:

- *the design phase* in which we structure the image;
- *the analysis phase* in which we scan the newly created image to detect possible vulnerabilities or security exposures.

To design images we should follow the principle of *"the less is better"*. Any extra feature, any extra layer that we add to our Docker image could represent an unnecessary vulnerability for our system. Our goal should be to create the minimal Docker image that can guarantee to run our application. For example, if our application can run standalone, without the need for additional libraries, a good choice would be to use as base image for our container *scratch*. Such image is used to create super minimal images, without adding any extra layer in our newly image. It can't be used for any application, just for the ones that contain a single binary, but it represents a good example of how we should design our images using as few layers and features as possible.

To analyse the newly created image we can use different tools, like *Docker Security Scanning* and *Clair* that will be described in the next paragraphs. The former one is proposed by Docker itself [51], while the latter is suggested by *Arbezzano* [40, Sec. 1.10].

Scan an image after a build can be a great way to avoid having vulnerabilities and exposures. For example it could happen to have an image that contains OpenSSL with the *heartbleed* vulnerability still present, the analysis phase could notify us about it in order to solve it before it becomes a problem.

4.6.1 Docker Security Scanning

Docker Security Scanning [51] is a tool developed by Docker Inc. and available as an add-on on both Docker Cloud and Docker Hub. It allows to scan private and official repositories, searching for known vulnerabilities. At the moment this tool is not open-source and it is available only to paid subscribers.

Docker Security Scanning uses *CVE* database as source of information to know the latest discovered vulnerabilities. *Common Vulnerabilities and Exposures* [52], or CVE, is a database for publicly known cybersecurity vulnerabilities, that indexes each vulnerability with an ID, a description and at least one public reference. Every time that a new vulnerability is inserted in the CVE database Docker Security Scanning updates its previous scan results. A scan is launched when a new Docker image is pushed to the Docker Hub or to the Docker Cloud.

Each scan analyse all the layers of a newly pushed image, identifying the components and creating an index for the hash of each one. Docker Security Scanning compares the hash of the component with the CVE database, reporting all the vulnerabilities and exposures found. At the end of the scan a summary is presented, detailing for each layer of a Docker image all of its components and for each component a list of the vulnerabilities found. Docker Security Scanning categorises every vulnerability as minor, major or critical, reporting its CVE code with the possibility to view its original report.

4.6.2 Clair

Clair [53] is an open-source tool developed by CoreOS. Like Docker Security Scanning, it allows to perform a static analysis of our Docker images, searching for known vulnerabilities. As opposed to Docker Security Scanning it is a free service.

Clair is written in Golang and it offers a set of HTTP APIs, that can be used by a client to pull, push and analyse images. Its implementation is based on the use of a database, where are stored all the known vulnerabilities and all the features present in our images. Clair uses different sources to download information about vulnerabilities, like Debian Security Tracker and Red Hat Security Data.

As described by Clair's documentation, there are four steps to follow to perform a static analysis of our application container:

1. when Clair is installed and then at regular intervals, information about known vulnerabilities are downloaded and stored inside the database;
2. a client can push a Docker image using Clair's API. This operation stores the image's features inside the database;
3. an analysis can be started always using Clair's API. Docker image's features are compared with the list of already known vulnerabilities, reporting at the end a summary of the vulnerabilities or exposures found;
4. Clair notify the system when there is an update for the information about a vulnerability.

4.7 Image Verification

Poisoned images, as seen in the previous section, can represent an important risk for the security when using Docker. In particular it is of primary importance to take some precautions before

using the `docker pull` command, making sure before that the image provided by the author is the same as what we get from the registry that we are using. In computer security this problem is better known as a *trust* problem, i.e. assuring both the integrity and the publisher of the data exchanged.

In most cases this problem is addressed using a mechanism of *cryptographic signing*, in which the data is signed with the *private key* of its author and verified with the *public key of the same*. Docker implements such mechanism via *Docker Content Trust*, which however is disabled by default and must be explicitly activated. Once activated it allow to verify both the integrity and the publisher of all the data received from a registry over any channel.

4.7.1 Docker Content Trust

Docker Content Trust [54] is a feature introduced with Docker Engine 1.8 that allows to verify the identity of the publisher of a Docker image. An image is signed by the Docker Engine with the private key of its publisher before being pushed to a remote registry, then when such image is pulled it is verified using the public key of the publisher to assure that it has not been tampered. The signature is referred only to a single TAG of the considered image. Docker Content Trust is disabled by default, but it can be activated using the following command inside a shell session: `export DOCKER_CONTENT_TRUST=1`. Its use does not influence regular Docker workflow, indeed it does not require special commands to be used. All the normal commands can be still used, with the exception that they only work with signed content.

Docker Content Trust is based on *The Update Framework* and *Notary* to provide both the integrity and the freshness of the content. The Update Framework or TUF is an open-source framework designed to make the update life-cycle safe. Notary, instead, is a utility for securely publishing and verifying content that is distributed over any insecure network. The entire mechanism of Docker Content Trust is based on the use of two different types of keys:

- *the Tagging Key* that is generated every time that a publisher creates a new repository and can be shared with anyone who can sign content for that repository;
- *the Offline Key* that is the source of trust for a repository and should never be shared with anyone. It is needed for creating a new repository and for rotating an existing Tagging key.

Docker also manages *Timestamp Keys* that are generated and stored on a remote server. These keys are used to guarantee the use of the most updated version of a particular content.

As described by its documentation, Docker Content Trust is particularly effective against three different types of attack:

- *Image Forgery*: if an attacker obtains to take a privileged network position, compromising a registry, he can still not be able to serve his tampered images to a user. Every Docker command, indeed, would fail in this situation, not being able to verify the content;
- *Replay Attacks*: if an attacker takes control of the network, providing to an user an outdated version of a Docker image, with the aim to exploit a known security vulnerability, he would still be stopped. Docker Content Trust uses the Timestamp key when publishing an image, ensuring that the user is receiving the most up to date content;
- *Key Compromise*: if an attacker compromises a Tagging key, Docker Content Trust can still guarantee the security of the system. It uses a hierarchy of keys in order to mitigate the loss of a key (exception made for the loss of the Offline key). In such a case the publisher can simply rotate the compromised key, removing it from the system.

4.8 ARP Spoofing Prevention

As described in the previous section, the default network configuration provided by Docker can be subject to ARP spoofing attacks. To mitigate this security threat there are mainly two possible solution, proposed by *Nyantec* [58] and that I will analyse in the following.

The first solution is to *drop NET_RAW capability*. In this way the containerised application would not be able to create *PF_PACKET* sockets, that are the ones used to receive or send packets at the device driver (OSI Layer 2) level, and so an attacker could not perform an ARP spoofing attack. The problem of such approach is that if on the one hand it is effective, on the other it also has many negative aspects due to the fact that many network tools (ping, traceroute, tcpdump, etc.) need NET_RAW capability to work.

The second and more accurate solution consists in the use of *ebtables*. The ebtables program can be used at host level and allow, among other things, to filter out ARP packets with incorrect sender protocol or hardware address, that is the case of an ARP spoofing attack. In general it is used to filter network traffic that pass through a Linux bridge at link layer and higher network layers. ebtables can be easily configured, for example we can consider a situation in which we want to drop all the ARP replies for a client registered with IP address 192.168.0.1 that do not come from the MAC address 00:1C:B3:de:d8:57. We can achieve such condition simply writing the rule:

```
ebtables -t nat -A PREROUTING -i eth0 -p arp --arp-ip-src 192.168.0.1
   --arp-mac-src ! 00:1C:B3:de:d8:57 -j DROP
```

4.9 Network Policies

Docker, as stated in the previous sections, is particularly used to build microservices. Such software development technique requires network policies to manage connections between services and also connections with the external network. In the last paragraph I have analysed solutions to prevent ARP spoofing attacks that act on the layer 2 of the OSI stack, in this paragraph I will describe how to enhance security on the higher layers.

iptables represent a well known solution for monolithic service, but they are not equally efficient with microservices. As described by *Adel Zaalouk* in a post on his blog [60], five hops of calls are needed from a containerised microservice to decide on whether a particular packet should be forwarded or not. Moreover, security policies defined by iptables are based on layers 3 and 4 of the OSI stack, while for a service it would be good to have policies that work also at the application layer.

Extended Berkeley Packet Filters or *eBPF* represent an appreciated solution to secure microservices. eBPF is an extension of the already existing *BPF*, that is a tool used to filter packets relying on filter-expressions that are parsed into byte-code to be then injected into the kernel in the form of native instructions. eBPF allows to reduce the steps needed to make a decision about a packet, thanks to the fact that it hooks into the kernel. In the context of containers it can be directly attached on the network namespace, intercepting and filtering all the calls quickly. There are different clients that can be used to define eBPF's security policies, in the next paragraph I analyse *Cilium*, a solution proposed by *Arbezzano* [40, Sec. 1.9] and that supports by default integration with Docker.

4.9.1 Cilium

Cilium [61] is an open-source project supported by Cisco and based on the use of eBPF. It operates at layers 3 and 4 of the OSI stack as well as at layer 7, providing protection also to higher level protocols like HTTP.

It allows to create security policies that are then compiled for eBPF en injected into the system. Docker uses labels to attach such policies to endpoints. An endpoint a Docker container that can be reached via the network. There are mainly three different types of policies that can be created:

- *Identity based Connectivity Access Control*: they operate at layer 3, indicating how the endpoints can communicate between each other relying on their labels. An example of policy that acts at layer 3, allowing the communication from endpoints with the label *role=A* to endpoints with the label *role=B*, is:

```
[{
  "labels": [{"key": "name", "value": "layer-3-rule"}],
  "endpointSelector": {"matchLabels": {"role":"B"}},
  "ingress": [{
      "fromEndpoints": [
        {"matchLabels":{"role":"A"}}
      ]
  }]
}]
```

- *Port Restrictions*: they work at layer 4, restricting the ports that can be used by an application to communicate. An example of policy that acts at layer 4, allowing all the endpoints with label *app=A* to emit packets using TCP on port 80, is:

```
[{
  "labels": [{"key": "name", "value": "layer-4-rule"}],
  "endpointSelector": {"matchLabels":{"app":"A"}},
  "egress": [{
      "toPorts": [
          {"ports":[ {"port": "80", "protocol": "TCP"}]}
      ]
  }]
}]
```

- *Application Level Access Control*: they work at layer 7, enforcing access control based on protocol calls like RPC or REST CRUD. An example of policy that acts at layer 7, allowing all the endpoints with label *app=A* to receive only *GET* requests to the URL */foo* on port 80 from all the endpoints with label *env=B*, is:

```
[{
  "labels": [{"key": "name", "value": "layer-7-rule"}],
  "endpointSelector": {"matchLabels": {"app": "A"}},
  "ingress": [{
    "fromEndpoints": [
      {"matchLabels": {"env": "B"}}
    ]},{
    "toPorts": [{
```

```
        "ports": [
          {"port": "80", "protocol": "TCP"}
        ],
        "rules": {
          "HTTP": [
            {
              "method": "GET",
              "path": "/foo"
            }
          ]
        }
      }]
    }]
  }]
```

A policy can be written like a JSON object, indicating the possible connections between endpoints. By default all the communications between endpoints are denied.

4.10 Docker Monitoring

As seen in the previous section, the security analysis of Docker can not stop at the static aspects, it should also cover the runtime activity of containers. Different tools can be used to achieve this goal, from the classical ones used for years to monitor services like *logging* to specific instruments that allow to audit and detect containers' activity like *Falco*.

4.10.1 Logging

Logging is a fundamental activity for every application. It is mainly used to provide information about the state just prior to an error in the system, helping to localise a problem in the application. Logs can also be fundamental at runtime, indeed they can be used to check if everything is working as expected.

The `docker logs` command can be used to show all the logs produced by a container, while the `docker logs service` command shows the ones produced by all the containers involved in a service. By default these commands show the output produced by a container and directed to *STDOUT* or *STDERR*.

Docker offers other mechanisms of logging called *logging drivers* [55]. Such mechanisms allow the integration of Docker with various supported log management tools, with the possibility to implement others using *logging driver plugins*. By default Docker uses *json-file* as logging driver, storing logs in JSON format on local disk. The *docker logs* shows only the logs stored in this way. The other logging drivers supported are:

- *none*: no logs are stored;
- *syslog*: logs are written to *syslog*, that must run on the host;
- *journald*: logs are written to *journald*, that must run on the host;
- *gelf*: logs are written to a *Graylog Extended Log Format (GELF) endpoint*;

- *fluentd*: logs are written to *fluentd*, that must run on the host;
- *awslogs*: logs are written to *Amazon CloudWatch Logs*;
- *splunk*: logs are written to *splunk* using the HTTP Event Collector;
- *etwlogs*: option only available on Windows, logs are written as *Event Tracing for Windows (ETW)*;
- *gcplogs*: logs are written to the *Google Cloud Platform (GCP) Logging*;
- *logentries*: logs are written to the *Rapid7 Logentries*.

Logging drivers can be configured in two different ways: at the daemon level, so all the containers will use the same configuration; at container level, so the specified configuration will be used only by a single container. In the first case the `/etc/docker/daemon.json` file must be modified, appending the option: `‘‘log-driver’’: ‘‘name_of_the_desired_logging_driver’’`. In the second case the flag `--log-driver name_of_the_desired_logging_driver` must be specified with the `docker run` command. Furthermore, logging drivers support two different modes for the delivery of a log message:

- *blocking*: it is the default mode, which blocks the operation of delivery until the driver is full;
- *non blocking*: it uses a an intermediate per-container ring buffer for consumption by driver to store the log messages.

4.10.2 Falco

Falco [56] is an open-source security tool developed by Sysdig and launched in 2016. It allows to carefully monitor the activity of a container simply installing it into the host's OS kernel. Thanks to this module all the system calls on a host can be monitored, regardless of whether these calls come from the OS, the container or the containerised application. Falco's developers describe their tool as a *"behavioural activity monitoring"*.

Falco is based on the detection of any behaviour that involves the use of a Linux system call. It can be configured to send an alert any time that a particular system call is executed, relying on its arguments and/or on the process that is calling it. As described on its documentation, examples of things that can be reported are: a container that is running a shell; a container that tries to read from sensitive files; a container that is running in privileged mode.

Sysdig's tool is based on a system of rules, where each rule is used to describe a behaviour or an event that must be monitored. Such rules are written as YAML files and forty default rules are provided by Falco when it is installed. A rule is composed by a description, the condition that must be monitored, the output text of the related alert and a level of priority.

Alerts can be configured in different ways:

- they can be delivered to the *STDERR*;
- they can be written on a file;
- they can be written to syslog;
- they can be piped to a spawned program, for example an email can be configured to be sent at each new alert.

4.11 Other Good Practices

There are many other good practices that can be followed for deploying Docker containers. Some of these can not be always used, depending heavily on the needs of the application we are developing. In this paragraph I have listed some of them:

- *Encrypt communication with the Docker daemon socket*: the Docker daemon runs by default via a non-networked Unix socket. In some situations it may be useful to run it using an HTTP socket, so that it could be reachable from the outside. In this case, as suggested by the Docker documentation [57], it is important to guarantee that all the communications towards the daemon are encrypted using TLS, preventing unauthorised clients from communicating with it. TLS can be enabled specifying the `tlsverify` flag when running the Docker daemon and client, together with the `tlscacert` flag to point to a trusted CA certificate. In this way the daemon will only accept clients authenticated by a certificate signed by that CA;

- *Defang SETUID and SETGID binaries*: set User ID (setuid) and Set Group ID (sgid) are two particular permission that can be set on binaries. Thanks to these permissions such binaries are always executed with the privileges of the owner or the group, so if a file is owned by the root user and it has for example the setuid bit set it will be always executed with root privileges. These types of files are very often used by attackers to gain privileges within the Docker system. A little trick proposed in the book "Using Docker. Developing and Deploying Software with Containers" [41, Sec. 13] is to avoid these types of file is to add in our Dockerfile a line in which such files are found and and their permissions are changed. For example to disable setuid rights the following line of code can be used:

  ```
  RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true
  ```

- *Set volumes to read-only*: a good technique for avoiding some kinds of kernel exploits and container breakouts is to set a shared volume between the host and a container as read-only. Although this solution is effective, it can often not be applied because many times the containerised applications need to modify files in attached volumes to work properly. A volume can be set as read-only appending `:ro` to the argument of the `-v` flag (that is used to with the `docker run` command to specify the volume that must be shared):

  ```
  docker run -v volume_name:/path/container:ro image_name
  ```

- *Not use a bridge interface*: as seen Docker configures by default a bridge interface for all the containers that are running on a host. Such configuration is however subject to attacks like ARP spoofing. In addition to the solutions already mentioned in this section, another possible remedy could be to use a different network configuration. For example a possible solution could consist in not using a bridge interface, delegating the host to route IP packets between the containers and internet. The drawback of this solution that it is not supported by default by Docker and it could not be so easy for a user to implement it. A user should indeed launch's a container with the `--net=none` flag and then he should set up by himself the network namespace and all the virtual interfaces;

- *Full virtualisation*: it might seem like a contradiction, but as pointed out by *Yasrab* [17, Sec. 5.2.13] the idea to use two layer of virtualisation could represent an optimal solution for certain kinds of security threats. As we have seen in the previous section, kernel exploits and container breakouts can be dangerous for a Docker system, allowing

an escalation from a container directly top the host. For this reasons the idea of adding a second layer of virtualisation could a winner in certain contexts. This solution can be achieved in two different ways:

- containers that belong to different users or that contain sensitive data could be segregated in separate virtual machines;
- Docker offers also the possibility to nest different Docker images together. For example *KVM* [59] is a generic container that can be used to launch virtual machines inside a Docker container. Such utility is often called *Docker-in-Docker*.

5 Practical Evaluation

As discussed in the two previous sections, using Docker without any precaution can lead to serious risks for the security of our system. For this reason, having a proactive approach is a necessary condition if we want to reduce the attack surface of Docker containers. In the previous section I have defined some best practices which can be followed for this purpose. In this section I will focus on two of the critical threats previously analysed, demonstrating how they can be easily mitigated just taking some precautions.

This section focuses on the practical use of the tools previously analysed, in order to try to prove how the ease of use of Docker is not affected by them.

5.1 Image Analysis

In this paragraph we will focus on the *analysis phase* of a Docker image. In the previous section I have introduced *Clair* as an open source tool that can be used to perform a static analysis of a Docker image, now we will see how it can be used in practice.

The Docker image that we will analyse is called *vaas-cve-2014-0160* [62], it is present on the Docker Hub, provided by the user *hmlio*. It is a simple image created for demonstration purposes, which is based on Debian Wheezy, using a version of *openssl* vulnerable to *heartbleed* (CVS-2014-0160).

As said we will use Clair to perform the image analysis. It can be used directly querying its API or through a third-party client. We will use the former method. There are many clients developed for this purpose that can be easily integrated with Clair. In particular we will use *Clair Scanner* [63], that is an open-source software written in Golang, that can be used together with Clair to speed up the analysis process.

5.1.1 Clair Installation

For this evaluation a MacBook Pro Retina Early 2013 with macOs 10.13.5 will be used. Clair itself can be launched as a container, together with a containerised database, that will be used to store information about the known vulnerabilities and the analysed image layers. Starting Clair from scratch requires a huge amount of time to fill up the database with the vulnerabilities information. For this reason the developers of Clair Scanner have created a Docker image of Clair that is updated daily, speeding up the launch process of a standalone Clair server. They also provide the image of a PostgreSQL database that can be used to store information about the Docker images that we want to analyse. To start these two containers we can use the following commands:

```
docker run -p 5432:5432 -d --name db arminc/clair-db:2018-06-01
docker run -p 6060:6060 --link db:postgres -d --name clair
   arminc/clair-local-scan:v2.0.1
```

The first command can be modified to use a newer version of the database, it is sufficient to change the tag of the used Docker image to the present day's date.

At this point we should have the two containers up and running, as we can verify using the command `docker container ls` as shown in Fig. 5:

```
bash-3.2$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS                              NAMES
737c85df70a7        arminc/clair-local-scan:v2.0.1   "/clair -config=/con…"   15 minutes ago      Up 15 minutes       0.0.0.0:6060->6060/tcp, 6061/tcp   clair
e3452c8959bf        arminc/clair-db:2017-09-18       "docker-entrypoint.s…"   17 minutes ago      Up 17 minutes       0.0.0.0:5432->5432/tcp             db
```

Figure 5: Clair successfully running

5.1.2 Use Of Clair Scanner

To install Clair Scanner we need to install first Go and *dep*, that is a dependencies manager for this programming language. The scanning tool can be downloaded directly from the GitHub page of the project and installed using the provided Makefile, with the command `make ensure && make build`.

Clair Scanner communicates directly with the Clair server previously launched to test our containers for vulnerabilities. It is also possible to specify a whitelist containing allowed vulnerabilities. To analyse a Docker image we can use the command `clair-scanner --ip OUR_LOCAL_IP IMAGE_NAME`. In our case it will be:

```
./clair-scanner --ip 172.22.116.88 hmlio/vaas-cve-2014-0160
```

The result of such command is showed in Fig. 6. It is possible to see how each layer of the Docker image is pushed to the Clair server, where it is analysed. In this case the image contains 234 known vulnerabilities. Such commands produces also a report of the vulnerabilities found, that will be analysed in the next paragraph. To specify a whitelist such command must be launched with the `--whitelist=""` option, specifying the path to the whitelist file. Such file can be a simple YAML file where the allowed vulnerabilities can be specified for each single image or for all the Docker images to analyse.

```
bash-3.2$ ./clair-scanner --ip 172.22.116.88 hmlio/vaas-cve-2014-0160
2018/07/17 15:29:52 [INFO] ▶ Start clair-scanner
2018/07/17 15:29:58 [INFO] ▶ Server listening on port 9279
2018/07/17 15:29:58 [INFO] ▶ Analyzing 3eb46b6b31ed387a272cce266cad568dc8eda5dc59f686094051af374ca3775b
2018/07/17 15:29:58 [INFO] ▶ Analyzing fc2576b894ea5efa7dacdef5976578ec60f855e204d7a3472711afcd40514bbd
2018/07/17 15:29:58 [INFO] ▶ Analyzing ab478682df00a9ed9ffa358272431cea4d21278d9b6782e1c0b2c493b8162a6e
2018/07/17 15:29:58 [INFO] ▶ Analyzing 7e416cf2c13694119acc04b332bf2c6acebb91a4fa612a77f2c8e43e119a6709
2018/07/17 15:29:58 [INFO] ▶ Analyzing b551a4c89e875f9ae919754d66fac1f2db73b540c0e790decb21175163dfa1fa
2018/07/17 15:29:58 [INFO] ▶ Analyzing 76da21d2fd4de3beb375ef4fa24aa49e19fb8766a3b7d02cb6abb5de59c1b2e3
2018/07/17 15:29:58 [INFO] ▶ Analyzing c7cea5e3d107bb3b23594556b3457245024e7851de58259b18f341f24d367a87
2018/07/17 15:29:58 [INFO] ▶ Analyzing dc0dc4e5e53c22012e053a1f7cae4779438b57a6dba100b8ccdaba949ecb46b9
2018/07/17 15:29:58 [INFO] ▶ Analyzing 0a16c6b3557987d375598534b9fed6fafa3f9efac95ba3c25346e40431111428
2018/07/17 15:29:58 [INFO] ▶ Analyzing 498191f40f6a78f8f6f46d05f7d4ddd4f9411510a2eda0b3719f4f3189b4ea9b
2018/07/17 15:29:58 [INFO] ▶ Analyzing 8fd1da6af1b69914373580d6c0c2e1d9c7f55edc3cf3cc4c1ea6afda405c9d7e
2018/07/17 15:29:58 [INFO] ▶ Analyzing ce2b1b6ae77df72b6ad973f8e7cc567bbb499202cf6d3fd9102b91349b3eff72
2018/07/17 15:29:58 [INFO] ▶ Analyzing 8b39a1d990e3eec1023bab848b87a7304be3c736b447f57e7539c81565348a2c
2018/07/17 15:29:58 [INFO] ▶ Analyzing 735de3a4178014e1b65317c593a1b0b3e56d043d1a3a7b0b4b536bc45c8c0f38
2018/07/17 15:29:58 [INFO] ▶ Analyzing 19d842b687dcd147d771bf15d229487e8e559cd63222ee8ea3812712875554a4
2018/07/17 15:29:58 [INFO] ▶ Analyzing 4608bf11f8ac48842bc76f5d66df35b8be643b88bd3c37691e30b60d64e1ca28
2018/07/17 15:29:58 [INFO] ▶ Analyzing 620693f7f6547388a2eeba046e10728c7a801cd4f7d2575aa9a6f9b610da4afd
2018/07/17 15:29:58 [INFO] ▶ Analyzing b42d200c43b48095c04f316e4c37c9716c3872f88bfa5c25d76a0fc742eeca5b
2018/07/17 15:29:58 [INFO] ▶ Analyzing c8e63206f1eb98430a62ae6567e84e0cffd846ac6eefd9cad648e24a75fd34e8
2018/07/17 15:29:58 [INFO] ▶ Analyzing d457e8b90e80a2fdc3531c2a3924073d044cc0581291bfd4bc25c85425d3b6d3
2018/07/17 15:29:58 [INFO] ▶ Analyzing ed1305204939d9793b44656716bb05b9b71080ea73300e32d60cc06fd58e3f68
2018/07/17 15:29:59 [INFO] ▶ Analyzing 782e6f1d69d29757c4d0b7020e073d46a078ae70bc72d1be255c41fe317db9c7
2018/07/17 15:29:59 [INFO] ▶ Analyzing d3077edf56d659d183609e62b8598a00ccb5dc0b54cfc2646349759958 25c55e
2018/07/17 15:29:59 [WARN] ▶ Image [hmlio/vaas-cve-2014-0160] contains 234 total vulnerabilities
2018/07/17 15:29:59 [ERRO] ▶ Image [hmlio/vaas-cve-2014-0160] contains 234 unapproved vulnerabilities
```

Figure 6: Clair Scanner Results

5.1.3 Results

As stated in the previous paragraph, Clair Scanner produces a report of the analysed image. Such report lists all the discovered vulnerabilities, ordering them by their CVE severity. For each vulnerability is reported its CVE identification code, the package where it was found together with the package version and a description of the vulnerability itself. The first vulnerabilities listed in the report of our analysed Docker image are showed in Fig. 7.

We analysed the Docker image *vaas-cve-2014-0160* because it contains a version of *openssl* vulnerable to *heartbleed*. Clair Scanner reports such vulnerability as can be seen in Fig. 8.

STATUS	CVE SEVERITY	PACKAGE NAME	PACKAGE VERSION	CVE DESCRIPTION
Unapproved	High CVE-2015-2059	libidn	1.25-2	The stringprep_utf8_to_ucs4 function in libin before 1.31, as used in jabberd2, allows context-dependent attackers to read system memory and possibly have other unspecified impact via invalid UTF-8 characters in a string, which triggers an out-of-bounds read. https://security-tracker.debian.org/tracker/CVE-2015-2059
Unapproved	High CVE-2014-9114	util-linux	2.20.1-5.3	Blkid in util-linux before 2.26rc-1 allows local users to execute arbitrary code. https://security-tracker.debian.org/tracker/CVE-2014-9114
Unapproved	High CVE-2016-0705	openssl	1.0.1e-2	Double free vulnerability in the dsa_priv_decode function in crypto/dsa/dsa_ameth.c in OpenSSL 1.0.1 before 1.0.1s and 1.0.2 before 1.0.2g allows remote attackers to cause a denial of service (memory corruption) or possibly have unspecified other impact via a malformed DSA private key. https://security-tracker.debian.org/tracker/CVE-2016-0705
Unapproved	High CVE-2017-12424	shadow	1:4.1.5.1-1	In shadow before 4.5, the newusers tool could be made to manipulate internal data structures in ways unintended by the authors. Malformed input may lead to crashes (with a buffer overflow or other memory corruption) or other unspecified behaviors. This crosses a privilege boundary in, for example, certain web-hosting environments in which a Control Panel allows an unprivileged user account to create subaccounts. https://security-tracker.debian.org/tracker/CVE-2017-12424
Unapproved	High CVE-2014-3512	openssl	1.0.1e-2	Multiple buffer overflows in crypto/srp/srp_lib.c in the SRP implementation in OpenSSL 1.0.1 before 1.0.1i allow remote attackers to cause a denial of service (application crash) or possibly have unspecified other impact via an invalid SRP (1) g, (2) A, or (3) B parameter. https://security-tracker.debian.org/tracker/CVE-2014-3512
Unapproved	High CVE-2016-0799	openssl	1.0.1e-2	The fmtstr function in crypto/bio/b_print.c in OpenSSL 1.0.1 before 1.0.1s and 1.0.2 before 1.0.2g improperly calculates string lengths, which allows remote attackers to cause a denial of service (overflow and out-of-bounds read) or possibly have unspecified other impact via a long string, as demonstrated by a large amount of ASN.1 data, a different vulnerability than CVE-2016-2842. https://security-tracker.debian.org/tracker/CVE-2016-0799
Unapproved	High CVE-2016-2182	openssl	1.0.1e-2	The BN_bn2dec function in crypto/bn/bn_print.c in OpenSSL before 1.1.0 does not properly validate division results, which allows remote attackers to cause a denial of service (out-of-bounds write and application crash) or possibly have unspecified other impact via unknown vectors. https://security-tracker.debian.org/tracker/CVE-2016-2182

Figure 7: Clair Scanner Report

Unapproved	Medium CVE-2014-0160	openssl	1.0.1e-2	The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug. https://security-tracker.debian.org/tracker/CVE-2014-0160

Figure 8: Clair Scanner Report

5.2 Container Monitoring

In the previous sections we have divided Docker's security concerns into two different types: static and dynamic. Regarding the latter we have seen how these are related to the runtime activity of a Docker container, for example a vulnerability not yet patched or a bug in our containerised application.

The use of log mechanisms or specific tools for monitoring container's activities can help in these contexts. In particular we have seen how Docker offers a native system for logging and how third-party software like Sysdig Falco can be used to actively monitor a Docker container. We will focus on this last application, showing how it can be used in practice to detect attacks.

5.2.1 Setup

For this practical evaluation we will use a virtual machine with Debian 9.4 and Linux Kernel 4.9.0. Falco's documentation can be found on its GitHub page [64]. It can be easily installed running the following command as root:

```
curl -s https://s3.amazonaws.com/download.draios.com/stable/install-falco
   | sudo bash
```

Then it can be launched simply using the *falco* command on the terminal.

It can be configured modifying the `/etc/falco/falco.yaml`. In particular we set it up in order to print all the events found on `/events.txt` and not on the *STDOUT*, adding the following code snippet to the configuration file:

```
syslog_output:
  enabled: false

file_output:
  enabled: true
  filename: /var/log/events.txt
```

As said in the first section of this research, despite a single Docker container can be used to run multiple processes, it is a good practice to execute just one process per container. In this way there is a better separation of concerns, moreover this allows to know exactly which process is running inside a certain container. This last statement can be used to enhance the security of our system. In most attacks, in fact, the attacker that takes control of a Docker container launches other processes inside the same container. Using Falco we can define a rule that can be used to warn in case a process different from the one initially defined is executed inside our container. We can append our rule to the file `/etc/falco/falco_rules.yaml`:

```
- rule: Unauthorised process
  desc: A process different from the defined one is spawned
  condition: spawned_process and container and container.image startswith
     nginx and not proc.name in (nginx)
  output: Unhautorized process (%proc.cmdline) running in (%container.id)
  priority: WARNING
```

spawned_process, *container*, *container.image startswith* and *proc.name in* are all macros defined by default in Falco. Such conditions detect a new process that is launched in a nginx Docker image and that is not nginx itself. Such rule also print out the name of the new process and the ID of the involved container.

5.2.2 Detection

As anticipated in the definition of the Falco's rule, the Docker image that we will use is nginx. It is one of the most popular Docker image on the Docker Hub and it can be used to create a web server, a load balancer, a reverse proxy, etc. In our case we will just use it to launch a process from its container's terminal. It can be launched using the command:

```
docker run -d -P --name falco_test nginx
```

We can verify that such container has been launched using the `docker ps` command as can be seen in Fig. 9, where it is also reported the ID of the container.

As said before, Falco can be simply launched using the `falco` command. It will automatically read the new configuration and the new defined rule from `/etc/falco/falco.yaml` and `/etc/falco/falco_rules.yaml`. At this point it will be sufficient to launch from the nginx's container any command to trigger Falco. In our case we will run the `ls` command via:

```
docker exec -it falco_test ls
```

We can see the generated warning checking the `/var/log/events.txt` file as showed in Fig. 10.

```
root@debian:/home/carmine# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
     STATUS              PORTS                    NAMES
8d326d1a82d9        nginx               "nginx -g 'daemon of…"   About an hour a
go   Up About an hour    0.0.0.0:32768->80/tcp   falco_test
```

Figure 9: nginx container successfully running

```
root@debian:/home/carmine# tail /var/log/events.txt
19:26:23.365218054: Warning Unhautorized process (ls ) running in (8d326d1a82d9)
```

Figure 10: Warning from Falco

5.3 Arp Spoofing Prevention

ARP spoofing attacks represent a serious threat for the security of Docker containers. As seen in the previous sections the default network configuration of Docker is prone to this type of attack. There are few countermeasures that can be taken to prevent such threat, including the use of *ebtables*.

In this subsection we will see a practical demonstration of how an ARP spoofing attack can be performed, following the guide provided by Philipp Bogaerts [36] and already described in 3.1.7. We will also see how using ebtables this attack can be mitigated.

5.3.1 Setup

For this practical evaluation we will use a virtual machine with Debian 9.4 and Linux Kernel 4.9.0. We will use the same containers described in 3.1.7, plus a fourth container that will run *TCPDUMP*. TCPDUMP [65] is a tool used to capture the packets in transit on a certain network interface, printing out a description of the their contents.

The following list shows and summarises the four used containers:

- two containers running the Docker image of *busybox*;
- a container based on Debian with *dSniff* installed. This is the container that will perform the attack. Its Dockerfile is the following:

  ```
  FROM debian
  RUN apt-get update && apt-get install -y dsniff
  ```

- a container based on Ubuntu with *TCPDUMP* installed, listening on the interface *eth0*. Its Dockerfile is the following:

  ```
  FROM ubuntu
  RUN apt-get update && apt-get install -y tcpdump
  CMD tcpdump -i -n eth0
  ```

5.3.2 Attack Execution

To perform the attack we will launch the first two containers, using the commands:

```
docker run -it --name client1 busybox
docker run -it --name client2 busybox
```

On both the containers we will use the command `ifconfig` to find out their IP and MAC addresses. In our case the first container (*client1*) has IP address *172.17.0.2* and MAC address *02:42:AC:11:00:02*, while the second container (*client2*) has IP address *172.17.0.3* and MAC address *02:42:AC:11:00:03*. From the second container we will ping the first container, simply using the command `ping 172.17.0.2`.

Then we will launch the third container with the command:

```
docker run -it --name arpspoofer arpspoof
```

Using the `ifconfig` command we can see that the IP address of such container is *172.17.0.4*, while its MAC address is *02:42:AC:11:00:04*. We will use then the `arpspoof` command to perform the ARP spoofing attack. In particular we use such command to get the traffic in both directions between *client1* and *client2*:

```
arpspoof -i eth0 -t 172.17.0.3 172.17.0.2 &> /dev/null &
arpspoof -i eth0 -t 172.17.0.2 172.17.0.3 &> /dev/null &
```

To verify the effective execution of this attack we will launch the fourth container using the command:

```
docker run -it --name tcpdumper --net=container:arpsoofer tcpdump
```

The `--net=container:arpsoofer` option is used in this case to share the network stack of the *arpspoofer* container with this one.

The correct execution of the attack can be seen in Fig. 11.

From such figure we can see how all the traffic between the fist two containers *client1* and *client2* go through the third one (*arpspoofer*). Moreover, from the first container it can be seen how its ARP cache is poisoned.

5.3.3 Attack Mitigation

Once the correct execution of this attack has been verified, we can see how the same can be mitigated using *ebtables*. ebtables can be simply installed on our host using APT:

```
sudo apt-get update
sudo apt-get install ebtables
```

ebtables are divided in three different tables: *filter*, *nat* and *broute*. In particular we will use the second one, which is the one used to filter packets at the link layer. We will define two policies, that will be used to drop all the ARP packets coming from the IP addresses of the first two containers but that do not match the real MAC addresses of such containers. We can define such rules using the commands:

```
ebtables -t nat -A PREROUTING -p arp --arp-ip-src 172.17.0.2 --arp-mac-src
   ! 02:42:AC:11:00:02 -j DROP
ebtables -t nat -A PREROUTING -p arp --arp-ip-src 172.17.0.3 --arp-mac-src
   ! 02:42:AC:11:00:03 -j DROP
```

We can verify the correct addition of such policies using the command `ebtables -t nat -L`, as showed in Fig. 12.

At this point we can proceed to re-execute the ARP spoofing attack, following the same steps previously analysed. In this case we can see how the attack is not successful, as it can be seen in Fig. 13.

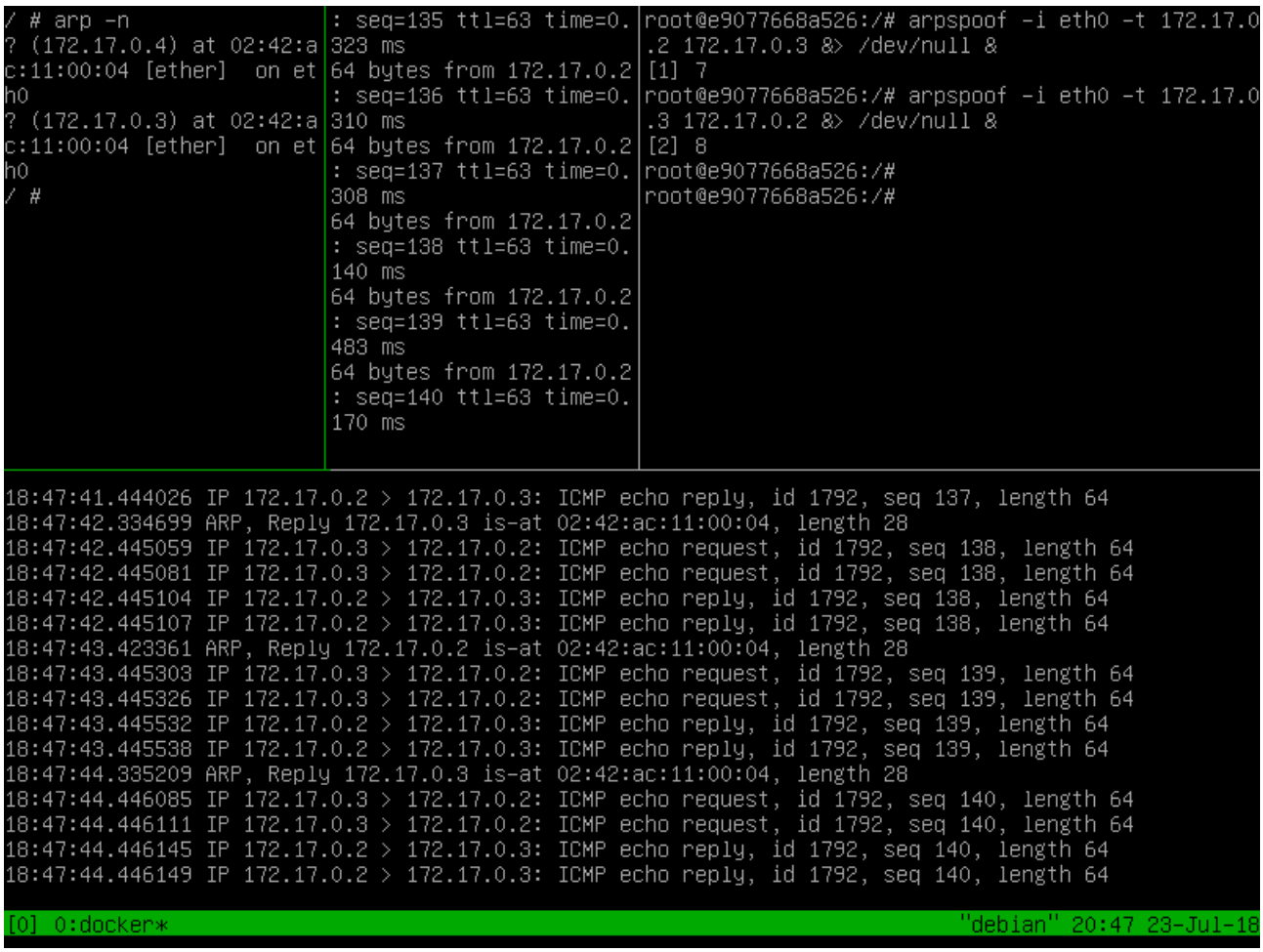

Figure 11: ARP spoofing performed

```
root@debian:/home/carmine# ebtables -t nat -L
Bridge table: nat

Bridge chain: PREROUTING, entries: 2, policy: ACCEPT
-p ARP --arp-ip-src 172.17.0.2 --arp-mac-src ! 2:42:ac:11:0:2 -j DROP
-p ARP --arp-ip-src 172.17.0.3 --arp-mac-src ! 2:42:ac:11:0:3 -j DROP

Bridge chain: OUTPUT, entries: 0, policy: ACCEPT

Bridge chain: POSTROUTING, entries: 0, policy: ACCEPT
```

Figure 12: ebtables policies

From this figure we can see how the ARP cache in the first container is not poisoned and also from the *tcpdumper* how the traffic between *client1* and *client2* does not go through the *arpspoofer* container.

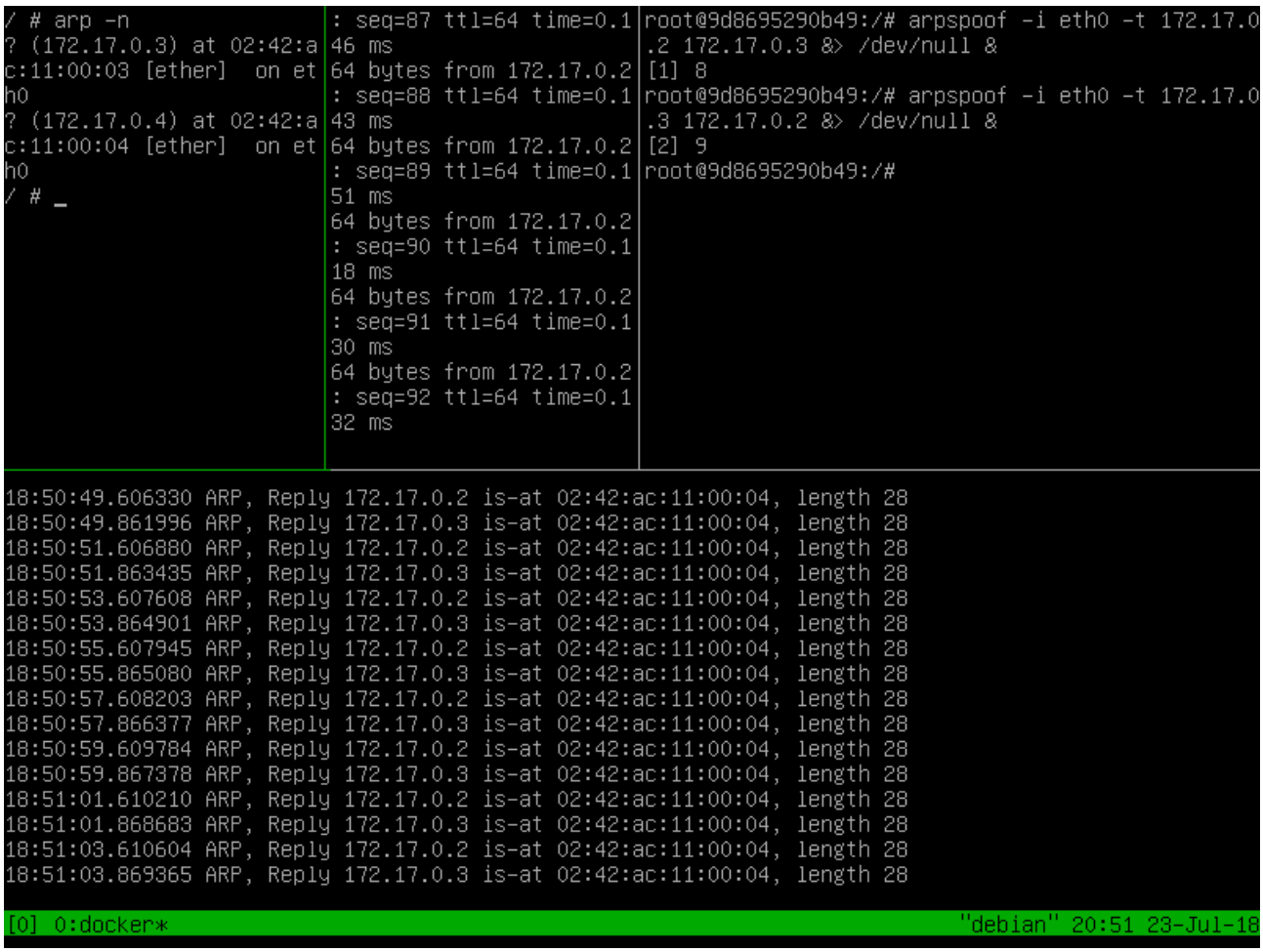

Figure 13: ARP spoofing mitigated

6 Conclusions

This research work focused on the security of Docker, in order to analyse this software program which, despite being still "young", has had a great success in the software industry. Several security threats were identified among those described in literature, providing where possible examples of real attacks to show how such threats are not only theoretical but also practical. These menaces range in different fields of the computer system security, from the security of the kernel to that of the network, from aspects strictly related to the use of Docker, such as the creation and use of images, to more general aspects concerning the development of network services, like the management of secrets. For each security threat one or more best practices were derived, with the aim of creating a workflow to follow when deploying a new Docker container. Such practices are based on the use of special configurations and ad-hoc policies or tools, in order to increase the security of a Docker environment. Two of these tools were also tested, demonstrating how their use can mitigate some of the analysed threats.

The main thesis of this research was that it is true that Docker has become popular thanks to its ease of use, which has sped up the development time of an application for many programmers, but it is also true that such simplicity hides behind itself security risks. In support of this thesis many security threats were analysed, showing how a simple `docker run` is not enough to guarantee the safety of the environment. At the same time it is important to remark the fact that Docker is a project with relatively few years of life and that its development is proceeding very quickly, solving in each version released several bugs and mitigating various security threats. It is also good to highlight the fact that Docker is an open-source project, so anyone can contribute to its code, making it safer and speeding up its development. Its popularity has also pushed several third-party developers to create specific tools for it, as the ones analysed in this research.

Docker has represented in these years a real "phenomenon" in the software industry, being quickly adopted by many companies, startups and independent programmers to sped up their development process. Its use carries with it various security risks that should not be ignored, but rather prevented following good practices such as those proposed in this research. The continuous release of new tools by Docker itself and also by third-party companies in order to address such security threats helps however to understand the attention placed on this software program. In particular if these new tools are able to improve the safety of Docker containers without affecting their ease of use, the use of Docker can only continue to grow.

References

[1] About Docker, https://www.docker.com/company

[2] Kubernetes, https://kubernetes.io/

[3] Virtualisation, https://en.wikipedia.org/wiki/Virtualization

[4] Thanh Bui, "Analysis of Docker Security", CoRR, January 2015, http://arxiv.org/abs/1501.02967

[5] LXC, https://en.wikipedia.org/wiki/LXC

[6] Introduction to Linux Containers, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers

[7] Introduction to Control Groups (Cgroups), https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

[8] Capabilities, http://man7.org/linux/man-pages/man7/capabilities.7.html

[9] Docker official site, https://www.docker.com/

[10] The future of Linux Containers, PyCon 2013, https://www.youtube.com/watch?v=wW9CAH9nSLs

[11] CI/CD in Docker, https://www.docker.com/use-cases/cicd

[12] Solomon Hikes, https://en.wikipedia.org/wiki/Solomon_Hykes

[13] Docker History, https://en.wikipedia.org/wiki/Docker_(software)#History

[14] Docker architecture, https://docs.docker.com/engine/docker-overview/#docker-architecture

[15] Docker objects, https://docs.docker.com/engine/docker-overview/#docker-objects

[16] The most common use cases of Docker containers and organisations, https://www.ibm.com/blogs/systems/the-most-common-use-cases-of-docker-containers-and-organizations/

[17] Robail Yasrab, "Mitigating Docker Security Issues", CoRR, April 2018, http://arxiv.org/abs/1804.05039

[18] Seven Docker security vulnerabilities and threats, https://sysdig.com/blog/7-docker-security-vulnerabilities/

[19] CVE-2016-5195, https://access.redhat.com/security/cve/cve-2016-5195

[20] MADVISE (2), http://man7.org/linux/man-pages/man2/madvise.2.html

[21] Demonstrating the Dirty Cow exploit, https://01.org/developerjourney/recipe/demonstrating-dirty-cow-exploit

[22] Denial-of-service attack, https://en.wikipedia.org/wiki/Denial-of-service_attack

[23] Limit a container's resources, https://docs.docker.com/config/containers/resource_constraints/

[24] Docker Container Breakout Proof-of-Concept Exploit, https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/

[25] Shocker, https://github.com/gabrtv/shocker

[26] Docker breakout exploit analysis, https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3

[27] Before you initiate a "docker pull", https://access.redhat.com/blogs/766093/posts/1976473

[28] Docker Image Insecurity, https://titanous.com/posts/docker-insecurity

[29] CVE-2014-9357, https://access.redhat.com/security/cve/cve-2014-9357

[30] J. Gummaraju, T. Desikan, Y. Turner, "Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities", BanyanOps, 2015, https://www.banyanops.com/blog/analyzing-docker-hub/

[31] What is Privileged Account Management?, https://www.coresecurity.com/blog/what-is-privileged-account-management

[32] Secret management using Docker containers, https://www.bbva.com/en/docker/

[33] IBM Data Science Experience, https://datascience.ibm.com/

[34] IBM Data Science Experience: Whole-Cluster Privilege Escalation Disclosure, https://wycd.net/posts/2017-02-21-ibm-whole-cluster-privilege-escalation-disclosure.html

[35] "dSniff" Wikipedia Page, https://en.wikipedia.org/wiki/DSniff

[36] ARP spoofing Docker containers, https://dockersec.blogspot.com/2017/01/arp-spoofing-docker-containers_26.html

[37] A. Raj MP, A.Kumar, S. J. Pai, A. Gopal, "Enhancing Security of Docker using Linux hardening techniques", 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), SJB Institute of Technology, Bengalaru, Karnataka (India), Jul 21-23, 2016, pp. 94-99

[38] J. Kabbe, "Security analysis of Docker containers in a production environment", Norwegian University of Science and Technology, June 2017

[39] T. Combe, A. Martin, R. Di Pietro, "To Docker or Not to Docker: A Security Perspective", IEEE Cloud Computing, September 2016, pp. 54-62, DOI 10.1109/MCC.2016.100

[40] Container security and immutability, https://gianarb.it/blog/container-security-immutability

[41] A. Mouat, “Image Provenance” in the book “Using Docker. Developing and Deploying Software with Containers” edited by B. Anderson, O’Reilly Media, December 2015, pp. 300-307

[42] ”Mandatory access control” Wikipedia Page, https://en.wikipedia.org/wiki/Mandatory_access_control

[43] bane, https://github.com/genuinetools/bane

[44] grsecurity Page, https://en.wikipedia.org/wiki/Grsecurity

[45] Hardened/PaX Quickstart, https://wiki.gentoo.org/wiki/Hardened/PaX_Quickstart

[46] Manage sensitive data with Docker secrets, https://docs.docker.com/engine/swarm/secrets/

[47] Docker in Production - Store Secrets using Hashicorp Vault, https://www.katacoda.com/courses/docker-production/vault-secrets

[48] What is Vault?, https://www.vaultproject.io/intro

[49] Consul, https://www.consul.io/

[50] Isolate containers with a user namespace, https://docs.docker.com/engine/security/userns-remap

[51] Docker Security Scanning, https://docs.docker.com/v17.12/docker-cloud/builds/image-scan/

[52] Common Vulnerabilities and Exposures, https://cve.mitre.org/

[53] Clair, https://github.com/coreos/clair

[54] Content trust in Docker, https://docs.docker.com/engine/security/trust/content_trust/

[55] Configure logging drivers, https://docs.docker.com/config/containers/logging/configure/

[56] Sysdig Falco, https://github.com/draios/falco

[57] Protect the Docker daemon socket, https://docs.docker.com/engine/security/https/

[58] Docker networking considered harmful, https://nyantec.com/en/2015/03/20/docker-networking-considered-harmful/

[59] Docker KVM simple container, https://github.com/BBVA/kvm

[60] eBPF, Microservices, Docker, and Cilium: From Novice to Seasoned, http://www.adelzaalouk.me/2017/security-bpf-docker-cillium/#security-policies-with-iptables

[61] Cilium, https://github.com/cilium/cilium

[62] hmlio/vaas-cve-2014-0160, https://hub.docker.com/r/hmlio/vaas-cve-2014-0160/

[63] Clair Scanner, https://github.com/arminc/clair-scanner

[64] Sysdig Falco Documentation, https://github.com/draios/falco/wiki

[65] Manpage of TCPDUMP, https://www.tcpdump.org/tcpdump_man.html