Security of Docker containers

Report for the Computer Security exam at the Politecnico di Torino

Carmine D'Amico (239540)
tutor: Antonio Lioy

? 2018

Contents

1 Introduction

2 Docker Overview

In computer science, the term *virtualisation* is referred to the creation of virtual computational resources [1]. 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 or 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 became popular thanks to its simple interface, which allows to create and manage containers in an easy way.

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 [2].

As said before, the hypervisor needs to run on the user's computer which is defined as *host machine*, while each virtual machines 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 results 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 look like separated processes that just share host's kernel and are more isolated from the host's system (Fig. 2).

Resources are provided by the host's OS together with the container engine. A 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 concern. It is better to connect multiple containers using user-defined networks and shared volumes.

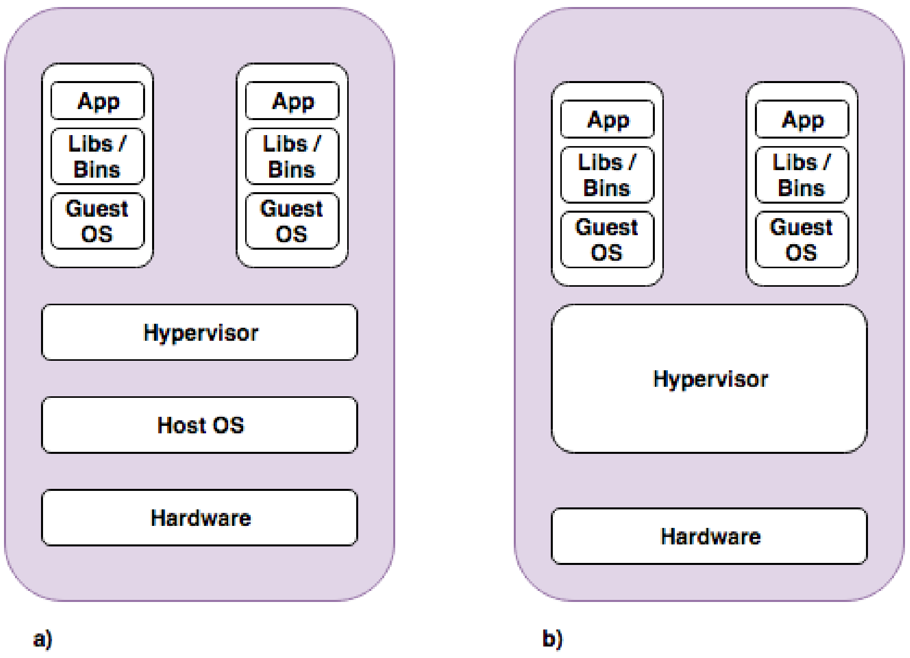

Figure 1: Architectural differences between (a) hosted hypervisor and (b) bare metal hypervisor

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 section allowing to better understand the behaviour of containers.

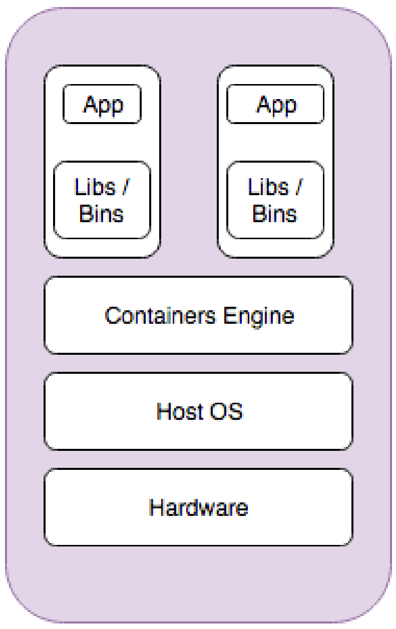

Figure 2: Architecture of a container-based virtualisation

2.3 LXC

LinuX Containers, better known as LXC, are an OS-level virtualisation technique created around 2008. They allow to run multiple isolated Linux instances (the *containers*), on top of a single *LXC host*, which shares its Linux kernel [3]. 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 these two components will be analysed, 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 six different type of environments are implemented [4]:

- **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 processes' resources. They gather, track and limit processes' usage of resources. It is possible to create and manage *cgroups* using high level code, assigning 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 [5]:

- **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.

2.4 Docker

As today, *Docker* represents the most used computer program for operating-system-level virtualisation (containerisation). It is developed by *Docker, Inc* [6] and it was introduced during the 2013's PyCon conference. During its presentation,

Docker was announced as the future of Linux Containers [7], indeed from its first releases it reiterated many concepts from them, such as *Namespaces* and *Cgroups*, 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 to not only build their code, but also test their code in any environment type and as often as possible to catch bugs early in the applications development life cycle [8].

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 [9] 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 of the milestones achieved by the program, from Wikipedia [10]:

- 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 [11] (Fig. 3):

- The server, which is a daemon process (called **dockerd**) running on the host's machine. It is in charge of create and manage Docker's objects.
- A set of interfaces conformed 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.

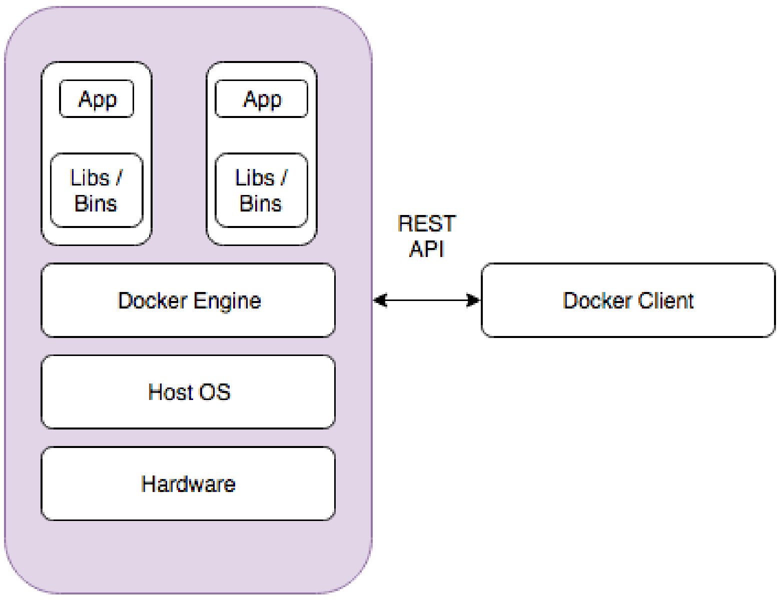

Figure 3: Docker's architecture

2.4.3 Docker's Components

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

- **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 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 a new layer is inserted (modifying the *Dockerfile*) and a container is rebuilt, 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, only the paid accounts can upload private images. These images take a standard name, that has the form "developer/repository". *Docker, Inc.* provides some official images, called simply "repository".

- **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 can be connected to the network or to a storage and it can be defined by its image or by the configurations indicated starting it. 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, ...

  The services that will run on a host can be configured by a special Docker file: *docker-compose.yml*. This file contains all the instructions that must be followed to run our application's services. For each service indicated it specifies where to pull the correspondent 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, ...).

- **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 for the IBM Infrastructure Blog stated [13] that the most common use cases for Docker are mainly three:

- *Integration test*: With virtual machines running integration test could take also days. The virtual machines should be configured and installed, then after the tests they should also be rebase back to their original state. On the contrary with Docker the configuration and the execution of a containers 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 practice 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 Issues" [14] to "To Docker or Not to Docker: A Security Perspective" written by Combe, Martin and Di Pietro [34], through Sysdig's list of the well known vulnerabilities of Docker [15]. Finally, I have divided the Docker attack surface in five macro areas:

- *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 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 simplicity 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 an impairment of the system. As well as the use of *outdated Docker images*, can lead to bugs in the system easily exploitable by an attacker.

- *Running Container*: Inside a Docker container there is anyway our running program. All the bugs and vulnerabilities that afflict our program can be used by an attacker to hit our system. For this reason, to cover all the attack vectors of our Docker containers, we should also *monitor the runtime activities of our containers.*

From these macro areas, 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 containers' operations, 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, so the attacker could read and write memory parts that belong to other containers.

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 section 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 [16].

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()* [17] with the MADV_DONTNEED parameter (such parameter indicates that 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 [18].

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 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 [19]. 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 containers' resources. However, as described on Docker's documentation [20], 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 [21], was fixed in 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 was 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. De facto 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* [22]. We will analyse such attack, with the aim to demonstrates how a Docker container could access some privileged file-system data. As said before, this kind of vulnerability was fixed with Docker 1.0, but it could still be interesting to understand how this type of attack worked [23].

Shocker takes advantage of the *CAP_DAC_READ_SEARCH* capability, that was granted by default to a superuser inside a Docker container. 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.

3.1.4 Poisoned Images

As said in 2.4.3, Docker images represent one of the most fundamental building blocks for a container. If an attacker obtains to make his images to run, the host and all its containers are seriously at risk.

Container images are claimed to be downloaded and verified by default by the system, thanks to the presence of a signed manifest inside the image. In such mechanism however Docker does not verify the image's checksum from the manifest. In this way an attacker could provide his *poisoned images* with some virus, together with a signed manifest.

Docker fetches and unpacks a container image in just one step without verification, using the command *docker pull*. Jonathan Rudenberg explained in a post [25] on his blog how the process of downloading a Docker image can be extremely insecure. Images are downloaded using HTTPS and then they go immediately through a processing pipeline inside the Docker daemon:

$$\text{decompress} \rightarrow \text{tarsum} \rightarrow \text{unpack}$$

This pipeline, even if it is functional, is insecure. The reason lies in the fact that the not trusted input is processed before its verification. A compromised image does not need to run to lead to malfunctions of the system. As stated by one blog post from Red Hat [24], an attacker can compromise the system even during the unpack step. For example an issue (now solved) exploited a tarball's capacity to perform directory traversal attacks, allowing compromised images to override parts of a host file system.

Practical Example

CVE-2014-9357 [26] is a 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 Static Images' Threats

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 our images 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 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*, ... These numbers grow up to the 40% taking in consideration also the images loaded by users [27].

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 are continuously created and destroyed. In both cases, sensitive information are needed, like API keys, database passwords, SSL/TLS keys, SSH keys, ... Compromising these information would compromise the entire system.

In a monolithic architecture the management of these information's is non trivial, having them stored in the system permanently with mechanisms for their renovation, like the "Privileged Accounts Managers" [28]. 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 [29]:

- 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.

Also 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 [30]. Such vulnerability was due to 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 many terabytes of sensible customer data.

As stated by the report of the vulnerability [31], an exploit of such threat was also quite easy, needing 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")
  ```

- 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 own IP addresses, IP tables, ... By default Docker provides connectivity between the containers creating a Virtual Ethernet Bridge, named *docker0*. At container creation time, Docker creates and connect 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 running a compromised host that belongs to the network. In the case of Docker, such conditions is satisfied if the attacker manages to compromises 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.

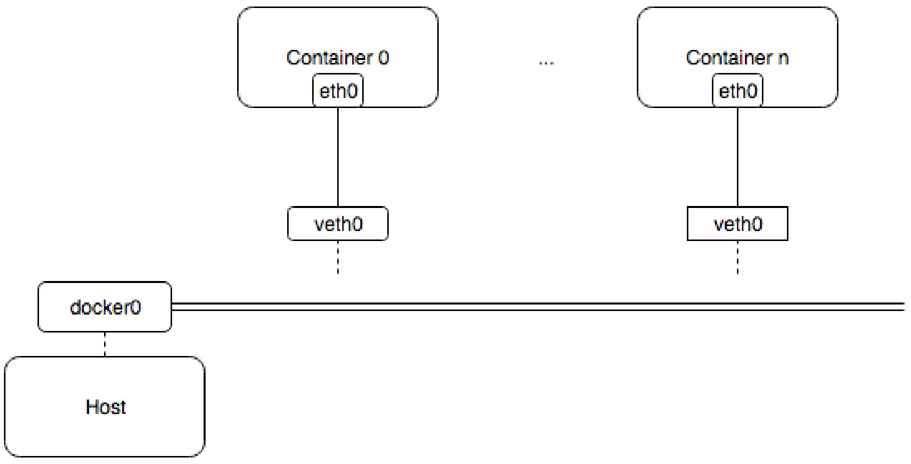

Figure 4: Docker default network model

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 [32].

Such type of experiment is described by Philipp Bogaerts in one of his blog post [33]. 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, to 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, ... 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 chapter, 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, 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 runtime, monitoring containers' activity . We can refer to them as *dynamic aspects* of Docker Security.

4 Best Practices For Docker Deployment

I started from the security threats analysed in the previous section to define some possible practices to follow during the deployment of Docker containers. In this section I lists these best practices. Not all of them refer to the security threats previously analysed, some are generic practices that can be useful in order to increase the security of the system.

The suggested tools are almost always either provided by Docker itself or open-source and free, I deliberately avoid proposing commercial and closed-source software because they are 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" [35] and to Adrian Mouat's "Using Docker. Developing and Deploying Software with Containers" [36] to find the best practices for security enhancement in Docker.

4.1 Mandatory Access Control

A *mandatory access control or MAC* system is a set of rules that define what a subject can or can not do in reference to a certain object. Usually the operating system plays the role of the policy administrator, while threads and processes are the subjects and files, directories, network ports, ... are the objects. When a subject tries to perform an action on an object the OS checks all the defined policies and decides if the action can be done or not. A user can not override or modify these policies, only the policy administrator can [37]. 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.

Docker's default profiles for these Linux kernel security modules are very generic, not so restrictive. They usually allow full access to the file system, network, and all capabilities of Docker containers. Moreover the general policy of these default profiles is to protect the host from the containers and not also containers from other containers [34]. 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.

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, ... are labelled. To achieve mandatory access control we have to write rules that control the access of a subject label on an object label. By default everything is deny, writing policies we can allow what we need.

Using 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. For example we can assign different security context to some host's file and Docker containers, in order to prevent the lasts from accessing to the firsts. Docker has to be launched with the *–selinux-enabled* flag to start it with.

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 many times other MAC tools are 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, which denies the access only to few filesystems on the host [2]. AppArmor profiles are simple text files where access control for capabilities and networking can be specified and rules for denying 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 objects like read, write, memory map as executable, file locking, creation of hard links, ... 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.

LiCShield

LiCShield [39] is an open-source tool that generates AppArmor profiles by following the behaviour of the Docker daemon during the execution of the build and run commands. The tool was presented at first only for LXC with the paper "Security hardening of Linux containers and their workloads" [40], then it was adapted to work also with Docker.

LiCShield works starting from a given Docker image, it traces the execution of the related container, collecting the information about the performed operations, their resources and required permissions. From the information retrieved LiCShield generates at the end a specific AppArmor profile for the analysed container.

Bane

Bane [38] 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", ...
- *Capabilities*, where a whitelist of the allowed capabilities is defined.
- *Network*, where allowed protocols and network policies are defined.

Bane installs directly 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 our 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 [41] 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* [42]: 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; ...

- 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, ...
- 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, ...
- 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 a normal 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.

4.3.1 Docker secrets

Docker secrets [43] 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 at rest 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 decoded 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 flag *–secret secret_name* with the Docker run command.

4.3.2 Vault

Vault [44] 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* [45].

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 can not 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 tree 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, they 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 provides its own ways [46], based on the use of cgroup, to manage how much memory, CPU, or block IO 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 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* (bytes), *k* (kilobytes), *m* (megabytes), *g* (gigabytes).

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 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 container is 1024.

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 IO 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 IO per second) with which a container can read to a device.

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* [47] of a Docker container to a less privileged user on the Docker host. 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 our Docker images is an important aspect in the use of Docker. Follow some basic rules during this process can be fundamental for our 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 our image.
- *The analysis phase* in which we scan the newly created image to detect possible vulnerabilities or security exposures.

To design our 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 our newly created image we can use different tools, in the next paragraphs I will introduce two of the most used. 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 imagine that contains OpenSSL with the heartbleed vulnerability still present, the analysis phase could notify us of such a problem in order to solve it.

4.6.1 Docker Security Scanning

Docker Security Scanning [48] 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 [49], 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 SHA of each one. Docker Security Scanning compares the SHA 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 [50] 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.

There are many clients that can be used with Clair, one of the most famous and used is *Clairctl* [51].

4.7 Image Verification

Most of the time Docker is used downloading third-party images and not creating your own images from scratch. As seen in the previous chapter one of the risk of this practice is to come across poisoned images. For this reason some precautions must be taken before running the *docker pull* command.

Red Hat suggests [25] different ways to protect ourselves while dealing with Docker images. The first and more general is to use only Docker images coming from trusted sources, for example only the official ones present on the Docker Hub which also have the guarantee of being scanned by Docker Security Scanning. Another way to improve security is to avoid the use of *docker pull*, separating the download and unpack/install steps. This can be done downloading Docker images using a security channel provided by a trusted source and then running *docker load* to load the image from the TAR archive downloaded. The last advice provided by Red Hat is to avoid any intentional or accidental access to the Docker Hub. In this way when you want to download a new Docker image you must provide explicitly its register of provenance, avoiding unintentional download due to some typos in a Dockerfile for example. This can be achieved blocking *index.docker.io* at the firewall-level or by modifying accordingly the */etc/hosts* file.

A possible solution for image verification is also provided directly by Docker with its *Docker Content Trust*, that I will analyse better in the next paragraph.

4.7.1 Docker Content Trust

Docker Content Trust [52] 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. 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 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 [55].

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, ...) 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 allow all the ARP reply for a client registered with IP address 192.168.0.1 and MAC address 00:1C:B3 on the eth0 interface and to deny all the other ARP replies on the same interface. We can achieve such condition simply writing two rules:

```
ebtables -A FORWARD -i eth0 -p arp --arp-ip-src 192.168.0.1 --arp-
    mac-src 00:1C:B3 -j ACCEPT
ebtables -A FORWARD -i eth0 -p arp -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 level 2 of the OSI stack, in this paragraph I will describe how to enhance security on the higher level.

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 [57], 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 levels 3 and 4 of the OSI stack, while for a service it would be good to have policies that work also at the application level.

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* that supports by default integration with Docker.

4.9.1 Cilium

Cilium [58] is an open-source project supported by Cisco and based on the use of eBPF. It operates at Layer 3/4 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 represents one or more Docker containers that share the same address and namespace. 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.
- *Port Restrictions*: they work at layer 4, restricting the ports that can be used by an application to communicate.
- *Application Level Access Control*: they work at layer 7, enforcing access control based on protocol calls like RPC or REST CRUD.

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* [53]. 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 [54] 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:

- *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 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 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* [56] is a generic container that can be used to launch virtual machines inside a Docker container. Such utility is often called *Docker-in-Docker*.

4.12 Summary

The relationship between the security threats and analysed during Section 2 and the best practices described in this section is illustrated in Table 1.

5 Experimental Evaluation

6 Conclusions

References

[1] ”Virtualisation” Wikipedia Page, https://en.wikipedia.org/wiki/Virtualization

[2] Thanh Bui, “Analysis of Docker Security”, January 2015

[3] ”LXC” Wikipedia Page, https://en.wikipedia.org/wiki/LXC

[4] 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

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

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

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

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

[9] ”Solomon Hikes” Wikipedia Page, https://en.wikipedia.org/wiki/Solomon_Hykes

[10] ”Docker History” Wikipedia Page, https://en.wikipedia.org/wiki/Docker_(software)#History

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

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

[13] 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/

[14] Robail Yasrab, “Mitigating Docker Security Issues”, April 2018

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

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

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

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

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

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

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

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

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

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

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

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

[27] G. Jayanth, T. Desikan, Y. Turner, "Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities", BanyanOps, 2015

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

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

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

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

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

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

[34] T. Combe, A. Martin, R. Di Pietro, "To Docker or Not to Docker: A Security Perspective", IEEE Computer Society, 2016

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

[36] A. Mouat, "Using Docker. Developing and Deploying Software with Containers", O'Reilly Media, 2015

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

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

[39] LiCShield, https://github.com/2m4t/LiCShield

[40] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, L. Foschini, "Security hardening of Linux containers and their workloads", 2015

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

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

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

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

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

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

[47] Isolate containers with a user namespace, https://docs.docker.com/engine/security/userns-remap/#about-remapping-and-subordinate-user-and-group-ids

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

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

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

[51] Clairctl, https://github.com/jgsqware/clairctl

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

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

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

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

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

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

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