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. OSs, 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 like the acceleration of the development cycle and the possibility to thicken applications on servers. *Docker* is an open source container technology, stepped into the limelight 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 that 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.

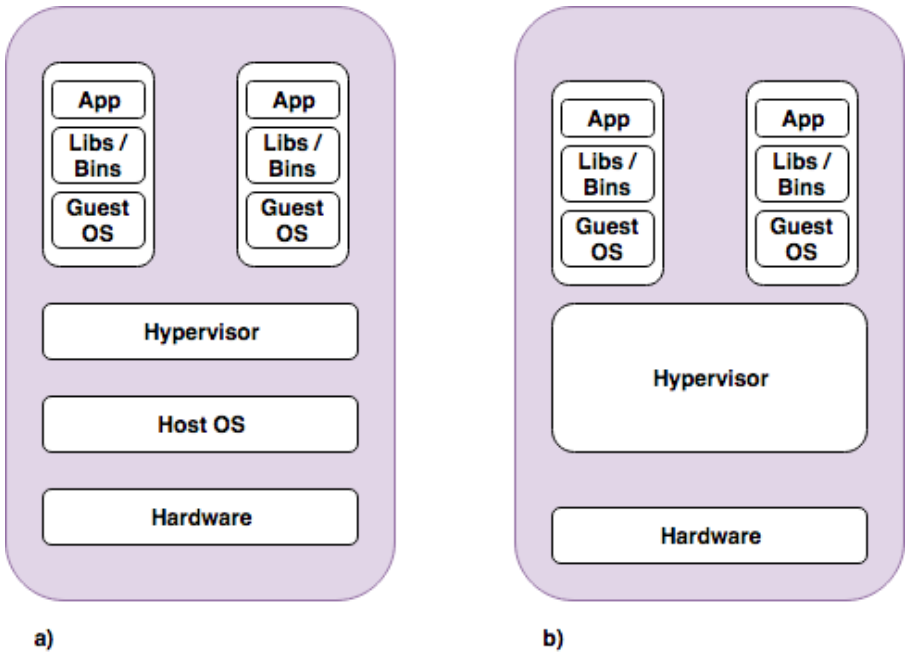

Figure 1: Architectural differences between (a) hosted hypervisor and (b) bare metal hypervisor

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.

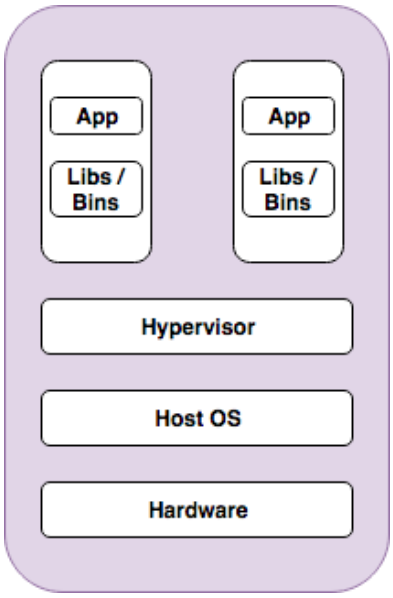

Figure 2: Architecture of a container-based virtualisation

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, ecc...

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, that 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 (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. [11]

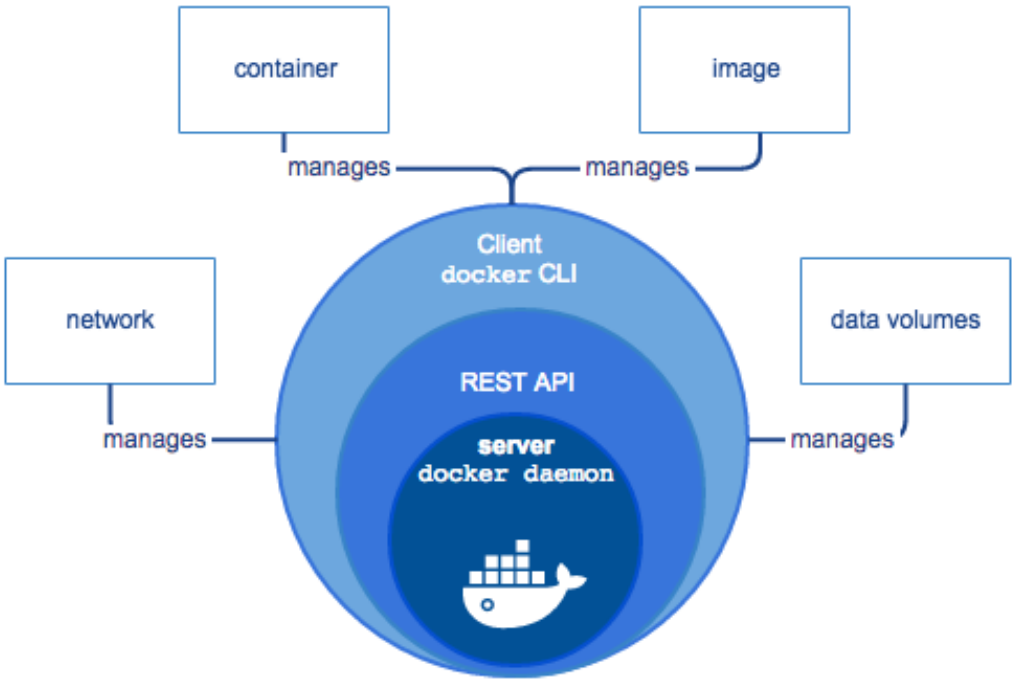

Figure 3: Docker's architecture

2.4.3 Docker's objects

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

- **IMAGES** are the basic components involved in the creation of a Docker's container, containing 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, that 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. They allow to run containers across different daemons. A series of daemons connected between them compose a *swarm* and they all communicate between them using Docker's REST API. A user can defines *swarm*'s configurations, like the number of *service*'s replica available. A *service* is seen from the external as a single application. [12]

3 Docker's security threats

4 Best practices for Docker deployment

5 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] CICD 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