

DevOps and Containers Security

Security and Monitoring in Docker Containers



JOSE MANUEL ORTEGA CANDEL

bpb

DevOps and

Containers Security

Security and Monitoring in

Docker Containers

by

Jose Manuel Ortega Candel



FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89423-532

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My Parents and Brothers

About the Author

Jose Manuel Ortega has been working as a software engineer and security researcher with a special focus on new technologies, open source, security, and testing. His career target has been to specialize in Python and DevOps security projects with Docker. Currently, he is working as a security tester engineer and his functions in the project are analysis and testing the security of applications both web and mobile environments.

He has collaborated with universities and with the official college of computer engineers presenting articles and holding some conferences. He has also been a speaker at various conferences both national and international and is very enthusiastic to learn about new technologies and loves to share his knowledge with the community.

Conferences and talks related with Python, Security, and Docker are available on his personal websites <http://jmortega.github.io> and <https://jmortegac.wixsite.com/conferences/conferences>.

About the Reviewers

Mitesh is a DevOps Evangelist. He is in love with the DevOps culture and concept. Continuous improvement is his motto in life with existing imperfection. He has recently authored a book named Agile, DevOps and Cloud Computing with Microsoft Azure (<https://www.amazon.com/Agile-DevOps-Computing-Microsoft-Hands/dp/9388511905>).

Ajay Bhaskar is a DevOps enthusiast and eager to learn new technologies related to automating application life cycle management. He loves to explore Docker. He has published an article Configuring Jenkins on Docker.

Acknowledgement

First and foremost, I would like to thank everyone at BPB Publications for giving me this opportunity to publish my book.

I would like to thank my teachers at the University for inspiring me to continuously learn in a world that is becoming increasingly complex.

Lastly, I would like to thank the reviewers and publishers for carrying out this project successfully.

—Jose Manuel Ortega Candel

Preface

In the last few years, the knowledge of DevOps tools in IT companies has increased due to the growth of specific technologies based on containers such as Docker and Kubernetes. Docker is an open source containerization tool that makes it easier to streamline product delivery and Kubernetes is a portable and extensible open source platform for managing workloads and services. The primary goal in the development of this book is to create a theory and practice mix that emphasizes on the core concepts of DevOps, Docker containers, and Kubernetes clustering from a security, monitoring and administration perspective. This book is helpful to learn the basic and advanced concepts of Docker containers from a security point of view. This book is divided into 11 chapters and provides a detailed description of the core concepts of DevOps tools and Docker containers.

Chapter 1 introduces DevOps methodologies and tools as a new movement that tries to improve the agility in the provision of services.

Chapter 2 introduces main Container platforms such as Docker Swarm, Kubernetes, and OpenShift that provide a common tooling for both development and operations teams.

Chapter 3 discusses how Docker manages images and containers, the main commands used for generating our images, and how we can reduce the attack surface minimizing the size of Docker images.

Chapter 4 covers topics such as security best practices and other aspects like Docker capabilities, which containers leverage in order to provide more features such as the privileged container.

Chapter 5 covers topics such as AppArmor and seccomp profiles that provide kernel-enhancement features in order to limit system calls. Also, we will review tools such as Docker Bench Security and Lynis that follow security best practices in the Docker environment.

Chapter 6 covers open source tools such as Clair with the quay.io repository and Anchore for discovering vulnerabilities in Docker images.

Chapter 7 discusses topics such as Docker Container threats and system attacks, which can make an impact in Docker applications like exploits that could target running containers. Also, we will review specific CVE in Docker images and how we can get details about specific vulnerability with the Vulners API.

Chapter 8 introduces Kubernetes Bench for the Security project as an application that checks whether Kubernetes is implemented securely by executing the controls documented in the CIS Kubernetes Benchmark guide.

Chapter 9 introduces the essential components of Docker networking, including how we can communicate and link Docker containers. Also, we will review other concepts like port mapping that Docker uses for exposing the TCP ports that provide services from the container to the host.

Chapter 10 talks about some of the open source tools available for Docker container monitoring such as cAdvisor, dive, and sysdigfalco.

Chapter 11 introduces some of the open source tools available for Docker container administration such as rancher and portainer.io.

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

1. Getting Started with DevOps

Structure

Objectives

What is DevOps?

DevOps methodologies

Management and planning

Development and building code

Continuous integration and testing

Automated deployment

Operations, ensuring the proper functioning in the production environment

Monitoring

Continuous Integration and Continuous Delivery

Software Delivery Pipeline

DevOps tools

DevOps and security

An introduction to DevSecOps

Conclusion

1. Container Platforms

Structure

Objectives

Docker containers

What is Docker?

Docker new features for container management

Docker architecture

Docker engine

Docker registry

Docker client

Testing Docker in the cloud

Container orchestration

Docker compose

Kubernetes

Kubernetes installation &key terms

Kubernetes cloud solutions

Docker swarm

Swarm in practice

OpenShift container platform

OpenShift as Platform as a Service

DevOps with OpenShift

OpenShift core items

Learning scenarios

Conclusion

1. Managing Containers and Docker Images

Structure

Objectives

Managing Docker images

Introducing Docker images

Docker layers

Image tags

Design considerations for Docker images

Dockerfile commands

What is a Dockerfile?

Building images from Dockerfile

Best practices writing Dockerfiles

Managing Docker containers

Search and execute a Docker image

Executing a container in background mode

Inspecting Docker containers

Optimizing Docker images

Docker's cache

Docker build optimization

Building an application with Node.js

Reducing image size with multistage

Reducing image size with alpine Linux

Distroless images

Conclusion

1. Getting Started with Docker Security

Structure

Objectives

Docker security principles

Docker daemon attack surface

Security best practices

Execution with a non-root user

Start containers in read-only mode

Disable setuid and setgid permissions

Verifying images with content trust

Resource limitation

Docker capabilities

Listing all capabilities

Add and drop capabilities

Disabling ping in a container

Adding capability for managing network

Execution of privileged containers

Docker content trust

Signing images mechanism

Secure download in Dockerfiles

Notary as a tool for managing images

Docker registry

What is a registry?

Docker registry in Docker hub

Creating Docker local registry

Conclusion

Questions

1. Docker Host Security

Structure

Objectives

Docker daemon security

Auditing files and directories

Kernel Linux security and SELinux

Apparmor and Seccomp profiles

Installing AppArmor on Ubuntu distributions

AppArmor in practice

AppArmorDocker-default profile

Run container without AppArmor profile

Defense in-depth

Run container with Seccomp profile

Reducing the container attack surface

Docker bench security

Execution examples with Docker bench security

Docker bench security source code

Auditing Docker host with Lynis and Dockscan

Auditing a Dockerfile

Dockscan for scanning Docker installations for security issues and vulnerabilities

Conclusion

Questions

1. Docker Image Security

Structure

Objectives

Docker hub repository

Docker security scanning

The Docker security scanning process

Open-source tools for vulnerability analysis

Continuous integration with Docker

CoreOS Clair

Dagda: the Docker security suite

OWASP dependency check

MicroScanner

Clair scanner and quay.io repository

Github repositories and Clair links

Quay.io image repository

Register in Quay.io

Analyzing Docker images with anchore engine and anchore cli

Starting Anchoreengine

Conclusion

Questions

1. Auditing and Analyzing Vulnerabilities in Docker Containers

Structure

Objectives

Docker containers threats and attacks

Dirty Cow Exploit (CVE-2016-5195)

Prevent DirtyCow with apparmor

Vulnerability jack in the box (CVE-2018-8115)

Most vulnerable packages

Analyzing vulnerabilities in Docker images

Security vulnerability classification

Alpine image vulnerability

CVE in Docker images

Vulnerable images in Docker hub

Getting CVE details with vulners API

Conclusion

Questions

1. Kubernetes Security

Structure

Objectives

Introducing Kubernetes security

Securing containers with Kubernetes

Configuring Kubernetes

Best security practices with Kubernetes

Firewall ports

Restrict the Docker pull command

API authorization mode and anonymous authentication

Kubernetes dashboard

Checking network policies

Pods security policies

Managing secrets

Kubernetes engine security

Handle security risks in Kubernetes

Increasing security using containers with Kubernetes

KubeBench security and vulnerabilities

CIS Benchmarks for Kubernetes with Kube-bench

Validating workers

Validating master

Kubernetes vulnerabilities

Kubernetes security projects

Kube-hunter

Kubesec

Kubectl plugins for managing Kubernetes

kubectl-trace

Kkubctl-debug

Ksniff

kubectl-dig

Rakkess

Conclusion

Questions

1. Docker Container Networking

Structure

Objectives

Introducing container network types

Types of Docker networks

Bridge mode

Host mode

Network managing in Docker

Docker networking

Containers communication and port mapping

Configure port forwarding between container and host

Exposing ports

Creating and managing Docker networks

Docker network commands

Bridge networks

Connect container to a network

Linking containers

Linking containers within the same host with --link

Environment variables

Conclusion

Questions

1. Docker Container Monitoring

Structure

Objectives

Container statistics, metrics and events

Log management

Stats in containers

Obtain metrics using docker inspect

Events in Docker containers

Others Docker container monitoring tool

Performance monitoring with cAdvisor

cAdvisor is a monitoring tool

Performance monitoring with Dive

Container monitoring with Sysdigfalco

Behavior monitoring

Wordpress container monitoring

Launching Sysdig container

Sysdig filters

Csysdig as a tool to analyze system calls

Conclusion

Questions

1. Docker Container Administration

Structure

Objectives

Introducing container administration

Container administration with rancher

Deploying Kubernetes using Rancher

Container administration with portainer.io

Deploying Portainer to Docker Swarm Cluster

Docker Swarm administration with Portainer

Conclusion

Questions

CHAPTER 1

Getting Started with DevOps

In this chapter, we will review the DevOps ecosystem as a new movement that tries to improve the agility in the provision of services. DevOps is more than a technology or a set of tools. It is a mentality that requires cultural evolution. The right people, processes and tools allow the lifecycle of applications to be faster and more predictable.

Structure

What is DevOps?

DevOps methodologies

Continuous integration and continuous delivery

DevOps tools

DevOps and security

Objectives

Understanding the concept of DevOps

Understanding DevOps methodologies

Understanding the concepts of continuous integration and continuous delivery and the software delivery pipeline

Knowing about DevOps tools

Understanding the concept of DevSecOps

What is DevOps?

In recent years, the evolution of technology has allowed us to achieve this communication between the development and operations teams, giving us the possibility of working with the infrastructure as a code, which makes it possible to work with processes that were previously manual or not very automated with the advantages of all the work that has been done in the development part to improve quality (test), collaborative work (version management), dependency management and integration with third-party products. These practices are oriented to reduce the time and effort in each of the development phases, managing to deliver code in production with greater speed and quality, reducing errors and limiting manual tasks that do not add value to the process.

DevOps is a software development methodology that seeks to optimize the delivery process as well as strengthen collaboration between the software development teams that build the solutions, and the operations teams responsible for these solutions are available in different environments.

The integration and collaboration of application developers (Dev) and those in charge of keeping them in production (Ops) offering important benefits:

Technical benefits:

Allows the implementation of continuous deployment strategies

Reduces risk and complexity

Cultural benefits:

Better communication, cohesion and motivation

Orientation to results, efficiency and quality of work

Professional development of team members

Creating a culture of shared responsibility, transparency and faster feedback is the basis of high-performance DevOps teams.

Business benefits:

Best time-to-market

More robust and stable operating environments

More resources to innovate (instead of correcting and maintaining)

Minimizes problem resolution time

Behind a simple definition, with an ambitious goal, we find some challenges:

DevOps is not an end itself, but a change in the culture of the organization, the tools used and the work procedures and methodologies.

It is necessary to know the strengths and weaknesses of the current software development cycle in order to define the best implementation strategy. This allows us to prioritize actions such as the implementation of tools and methodological changes.

It is very important to define the indicators that allow evaluating the effectiveness of the different actions: on the one hand to correct those that are not giving the expected result, and on the other to consolidate the cultural change in the organization.

With the advent of agile development methodologies and the needs of continuous integration and delivery (CI , continuous integration and CD , continuous delivery) there is a new organizational trend called DevOps, which, in short, aims to combine profiles into a single team very separated in more traditional organizations such as developers and operations teams, all with the final goal of deploying in productive environments more regularly.

Making new deliveries of the software on a regular basis (weekly, daily or even several times a day) is achieved to provide the process of the production step of more security or stability and more efficiency.

According to the DevOps state study, it is proven that organizations that use agile development methodologies and DevOps philosophy in their organization deploy up to 46 times more frequently than more traditional organizations, with failure recovery times 96 times faster and with a failure rate changes 5 times less than more traditional organizations, not so focused on performance. Deployments are made in production much more often (on-demand, several times a day) with a lead time for changes of less than one hour.

The term DevOps (Development + Operations) postulates that in business software, the line that divided the development of operations has been deleted. When new development methodologies (such as agile software development) are adopted in a traditional organization with separate departments for Development, Operations, Quality Control and Implementation, where before there was no deep need for integration between these IT departments, they now require close a multi-departmental collaboration.

DevOps involves the tasks automation of creating a job for development, but also the systematization of tests, deployment and configuration tasks related to it, all in an environment of agile development. Specifically, DevOps comprises the following 7 aspects:

Automation of tasks related to development: You do not have to remember commands to do all kinds of things (installation of libraries or

configuration of a machine), but there are scripts that homogenize and automate specific tasks in development phase.

Virtualization: use of virtual resources for storage, publication and, in general, all the steps of software development and deployment.

Servers provisioning: the virtual servers to which they are deployed must be prepared with all the necessary tools to publish the application.

Management of configurations: the management of the configurations of the servers and the orders for provisioning must be controlled by a version management system that allows testing and control the environment in which the software is running.

Deployment in the Cloud: publication of applications in virtual servers. The Cloud is a key environment that facilitates the development of DevOps since it provides this methodology with the speed and automation capacity necessary to make innovation and model change possible.

Software life cycle: the life cycle of an application includes the definition of the different phases in the life of an application, from design phase to support phase, going through the development phase.

Continuous deployment: the life cycle of an application must be linked to agile development cycles in which each new feature is introduced as soon as it is ready and tested; Continuous deployment implies continuous integration of new features and fixes, both in software and hardware.

DevOps proposes an agile and collaborative interaction between developers and operations team, from the traditional perspective with marked segregation of functions, through the inclusion of mechanisms that give greater dynamism to the delivery of services without neglecting control from the beginning of the project until production control. To achieve its objective, DevOps is based on principles such as Continuous Integration, Continuous Delivery and Continuous Deployment.

Figure 1.1: DevOps as an intersection between development, operations and QA

DevOps establishes an intersection between development, operations and Quality, but is not governed by a standard framework of practices and allows a much more flexible interpretation to the extent that each organization wants to put it into practice, according to its structure and circumstances.

The term DevOps refers more than just implementations of software: is a set of processes and methods to think about communication and collaboration between the departments mentioned above. Companies that have very frequent software deliveries may require a DevOps awareness or orientation. The adoption of DevOps is being driven by factors such as:

The use of agile development processes and other methodologies.

The increase of a higher rate of production versions by the interested application and business units.

Wide availability of virtualization in the cloud infrastructure of internal and external suppliers.

Increased use of data automation and configuration management tools.

The following points could be considered fundamental for adopting a DevOps methodology by an organization.:

Use of agile methodologies -agile methodologies such as Scrum allows developers using iterative and incremental approaches using multidisciplinary teams and try to deliver products with the highest possible value to the client in the shortest possible time. This methodology can be complemented with other tools like Kanban as a tool to manage development tasks oriented for visualizing the tasks workflow, work in progress and completed tasks.

Other methodologies such as Extreme Programming (XP) has the great advantage of organized and planned programming so that there are no errors throughout the process. They are usually used for the execution of short-term projects. It is considered a light methodology and focuses on cost savings, unit tests, integration of the whole system on a frequent basis, pair programming, simple design and frequent deliveries of software that works.

Testing methodologies such as BDD (Behaviour Driven Development), TDD (Test Driven Development) and ATDD (Acceptance Test Driven Development) have acquired great importance in software development to help an organization to test and improve the efficiency of development successfully. These methodologies can be complemented with other techniques like white box and black box tests for test performing.

Using a microservices architecture is one of the best ways to solve the problems inherent in monolithic systems. This type of architecture improves the assignment of responsibilities in the development teams and facilitates the encapsulation in Docker containers, reducing the effort and risk of managing the dependencies of the application, improving the management of updates and providing functionalities such as load balancing, high availability and service discovery. Containers technology has the advantage of sharing operating system and isolates applications by adding a layer of protection between them.

Use of good practices - these good practices include activities aimed at correctly implementing DevOps and refining the problems that may arise in adapting to the organization.

Record all incidents: Each incident must be reflected in a tool for further processing.

Guarantee repeatability: Every operation should be automated as much as possible, providing automatic mechanisms also for the rollback to the previous state of a change.

Test everything: Every change should be tested, if possible, in an automatic way. To automate are the integration / deployment / continuous delivery systems that have already been mentioned.

Monitor and audit what is necessary: Using tools for tracking applications behavior, as well as incidents in the logs, is very useful for the development team to fix problems. In addition, there must always be a person responsible for each change in the system, and generic accounts must be avoided, each user must carry out operations in an identifiable way.

DevOps methodologies

Currently, DevOps can be defined as an infinity symbol or a circle that defines the different areas and phases that comprise it:

Planning

Developing (build phase)

Continuous integration and testing

Deployment

Operation

Monitoring (continuous feedback)

Figure 1.2: DevOps processes

It is important to understand that it is one of the multiple representations, not the definitive canon. Having fully valid simplifications in the form of four main phases, or detailed decompositions of each of them.

Another essential idea to internalize is that it is the definition of an iterative flow so that different processes can be included in different phases in an organic and superimposed way, always adjusting to the fundamental concepts of value and continuous improvement.

Now, I will look at each phase in more detail, allowing me a very usual license in the DevOps processes, which is to use the Scrum framework as a working methodology to make explanations easier.

Management and planning

Every project needs a vision that indicates to the participants the reason and the goal of the work to be done; defining a minimum set of functionalities that allow to provide functional value in each iteration, the acceptance criteria to be met and the definition of done; for each one of the phases and in the whole of the project.

This is constituted as a living product stack, which is continuously supporting a process of gardening, from a business point of view, which feeds the different phases of development and operations; and that addresses changes and developments according to a process of continuous improvement based on early and continuous feedback.

In this phase, it is essential that business and management teams are formed in the tools and metrics designed so that they have a true and enough visibility of the development of the project.

Development and building code

This phase is where it is built the application, designing infrastructure, automating processes, defining tests or implementing security. It is where the most important effort is being made in the automation of repetitive or complex actions; and that it should be one of the first steps to scale to implement DevOps in an organization.

If I had to summarize in a single word the most important concept of this phase, this would be evidence. Either in a management application, operations with data or the deployment of virtual infrastructure; I will always work in code - either with a programming or scripting language; which must be stored in a code manager that allows basic operations such as historical, branches, versioning, etc.

But this is not enough, and each piece built must include its own automated tests. That is, the mechanisms that the system itself can make sure that what we have done is right does not fail, does not fail elsewhere, meets the acceptance criteria and points of early errors that arise in all development.

First, I store the code in a control version manager like Git or Bitbucket in order to have versioning and rollback; then including automated testing. Finally, we arrive at the orientation of what was built towards the following phases, including the transformation of the workflow itself.

Continuous integration and testing

Although in this phase and the previous one most of the authors focus on a development point of view, the arrival of DevOps and the concepts of Infrastructure as a code, make IT also a full participant of this phase.

The continuous integration is to automate the mechanism of review, validation, testing and alerts of the value built in the iterations, from a global point of view. That is, my unique functionality or feature, which I have built in my development environment, together with the automatic tests that ensure its proper functioning, are published in a service that integrates it with the rest of the application.

Regarding continuous testing, by launching all the tests included in each functionality, plus the integration tests of the whole application, plus the functional tests, plus the acceptance tests, plus the analysis of the quality of the code, plus the regression tests. In this way, you can be sure that your application is still working correctly.

And if something goes wrong, the early warning will jump, indicating in what piece and in which line it is breaking my system. So, the closer I get to the moment of initiating the critical path of deployment, the quieter I will be because more evidence includes my work.

Automated deployment

Deploying, in classical organizations, has always been a difficult task. Two roles (Dev and IT) with divergent objectives and interests are in a battle of isolation and mutual suspicion to publish the application in different work environments: development, integration, testing, pre-production and production.

As in any chain, it is easy to break through the weakest link, and the more steps there are in the deployment processes, the more possibilities of human failure are added. Thus, DevOps promotes the automation of deployments through tools and scripts, with the goal of having the entire process resolved with an approval button or, ideally, the activation of a feature.

For each environment, it is important to perform and provide the different types of tests (such as performance, resistance, functional, safety or UX

tests) in addition to managing the configuration of the different environments.

The most critical and difficult in this phase, more than known and adopted in the IT environment, is the arrival of the cloud concept with its infrastructure capabilities as a code, which forces a change in the paradigm of infrastructure management.

Operations, ensuring the proper functioning in the production environment

It is a minority the applications that are put into production and do not require constant work in its optimization, evolution, or support. But, in addition, you must take aware of all operations related to its operation that must be carried out continuously throughout the life of the software.

In this way you will have the adjustment of the resources according to the demand or the characteristics regarding the growth of the applications; the dynamic modification of the infrastructure due to security, performance and availability; or the optimization of processes and procedures that require changes in the context of execution and exploitation.

In this phase, it will apply the adoption of the concept of cloud - be it public, private or hybrid - where operations can exploit the capabilities of scalability, persistence, availability, transformation, resilience and security offered by this type of platform.

Monitoring

This last phase of a DevOps process is a permanent phase and applies to the entire cycle. It is where you are going to define the measures that will be monitoring to control the health status of the applications and their infrastructure.

But not everything is technology, and in this phase, the continuous feedback of all the areas and levels of the DevOps cycle will be consolidated to be included in the next iteration during the Plan phase, or immediately with specific corrections.

The goal of this phase is monitoring and measuring everything that can give you an overview of the current project status, including all the dependencies, but with capacities to go down to the singularity for observing the operation of a particular piece carefully.

Continuous Integration and Continuous Delivery

DevOps manages principles that are part of the collaborative structure and are used throughout the development and deployment of applications.

The principles bywhich DevOps operates are the following:

Continuous Integration

Continuous Delivery

Continuous Deployment

Continuous delivery focuses on making the development of a product always in a state of delivery throughout its life cycle. Continuous delivery improves efficiency and adjusts the planning and budget of the software delivery process, making it cheaper and less risky to release new versions of the software to the customer.

The implementation of continuous delivery means creating multiple feedback loops to ensure that the software is delivered to the customer more quickly. One of the requirements of continuous delivery is that all people, developers, system technicians, QA and operations collaborate effectively throughout the delivery process.

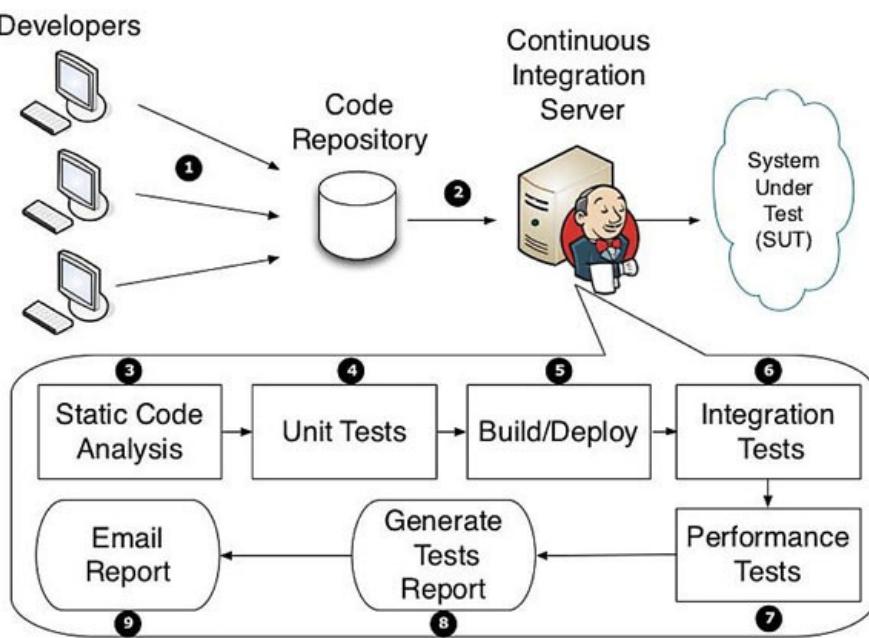


Figure 1.3: Continuous Delivery process

Continuous integration is a development practice by which developers routinely merge their code into the central branch (also known as master or trunk) into a version control system - ideally, several times per day. Each change triggers a set of quick tests to discover possible errors, which the developers must solve immediately.

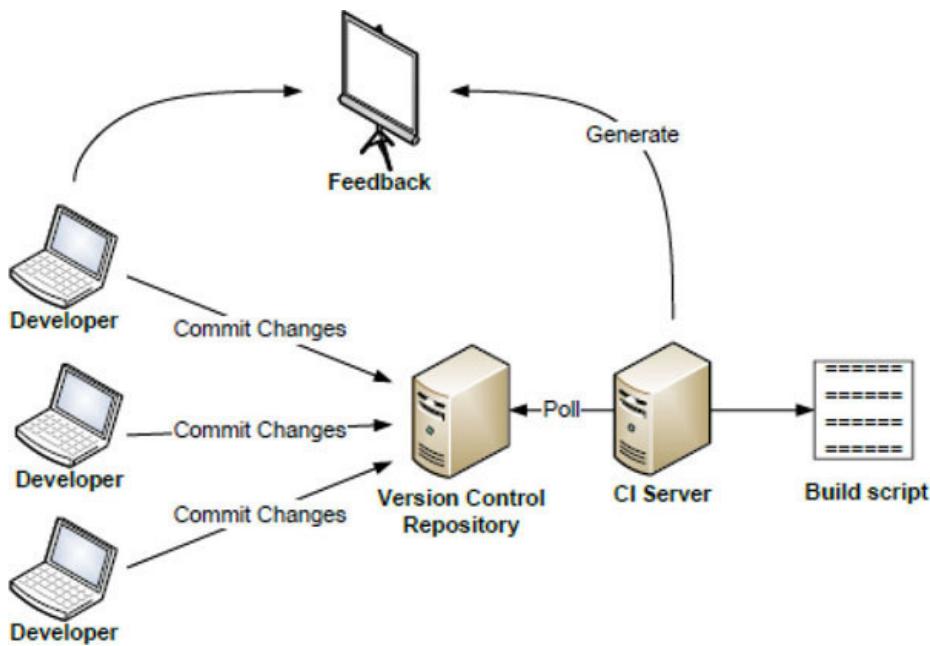


Figure 1.4: Continuous Integration process

These practices, which are a critical part of continuous integration and delivery, also require test automation and version control. Functional validations, such as performance and usability tests, give the team the opportunity to detect problems introduced by the changes as soon as possible and solve them immediately.

The objective of the integration and continuous delivery is to make the process of releasing the changes to the final client technically simple, that is, a routine and boring process. At this point, the IT team can devote more time to planning tasks and proactive strategies that can produce even more value to the company.

One of the greatest advantages in speeding up the delivery of applications through the development of the complete life cycle is that they can be developed iteratively and then delivered to production on demand by the client.

The Software Delivery Channel is made up of all the processes that speed up the generation of value to the client, minimizing risks and blockages. Here are the phases of this software delivery channel:

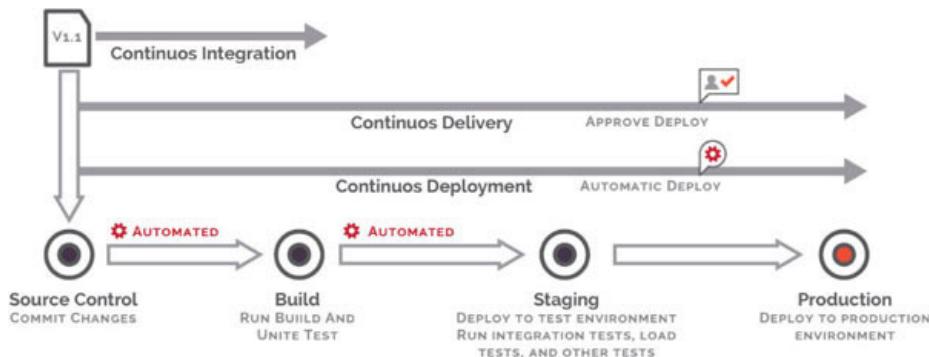


Figure 1.5: Software Delivery Pipeline

Continuous Integration

Continuous Integration is the way in which the software development team integrates its partial or total work, in a certain time established by the work team. It requires automation tools that are unique to the entire team of developers. These tools help to integrate into continuous form parts of code that are validated by automatic tests, which makes the work of the development team more efficient since it allows detecting failures in the early stages of the development cycle.

Continuous integration is originated under the extreme programming methodology and is a software development practice that requires the periodic integration of code changes into a shared repository. This strategy involves doing integration as often as possible. Although it sounds contradictory, this practice allows small and simple integrations to be made constantly, reducing time and increasing speed. In order to have a continuous integration process, several useful steps can be followed:

Have a code repository in which the development is centralized. Each developer works on small tasks, and when each task has finished the changes to the central line of the repository are included.

Start a process of compilation and testing in an automated way, that proves that the changes and additions made are correct and have not altered any part of the software. For this to work properly, it is essential that there is a good set of tests that can be trusted.

Execute this process several times a day, paying attention to the reported errors, which become a priority until they disappear. With this, it is always possible to have the latest functional version of the project status on the mainline, a version that is updated several times a day.

Continuous Delivery

Once the integration is achieved, you must continue with the cycle to perform the testing that, if satisfactory, allows the application to be deployed. In this stage, all changes in the code are implemented in order to generate an application that can be tested in production.

Continuous delivery represents a step beyond continuous integration. According to Martin Fowler, the continuous delivery is to build the software in such a way that it is always ready to go into production, taking aware of the following features:

It is deployable throughout its life cycle.

The teams have prioritized this drop-down feature at any time on the construction of new features.

Can give fast and automated feedback.

Can be deployed in any version and environment (development, testing, production), on-demand.

In continuous delivery (CD), the integrated code (CI) is automatically tested through many environments throughout the process to reach the preproduction phase, where it is ready to be implemented definitively. The interaction between CI and CD is called CI/CD.

Continuous deployment

Continuous deployment is the next step to continuous delivery. The continuous deployment is a practice that allows bringing the results of a development process to an environment similar where the functional tests can be given at full scale. The objective is to detect problems in production as quickly as possible. It is the early moment in which the user interacts with the application, reviews their requirements and can go back in the development.

The continuous deployment requires a configuration of the work environment, which allows an effective functioning of the candidate versions by the users. It begins with a pre-configuration during the entire development process and a final configuration before the candidate version is finished.

DevOps tools

Due to the growing ecosystem of DevOps-related tools, a quick review of the different categories will be carried out, describing in some detail some of the most widely used tools and simply naming some others.

The objective of the DevOps collaboration is to update and maintain stable production systems. The tools serve as support to overcome the development and operations gap, and they can be used in the development stage exclusively as Dev, and also in the deployment stage as Ops. For both Dev and Ops, there are tools with software, code, scripts or scripts that help to control the environment of development and deployment of applications efficiently.

The DevOps tools are part of the strategy of good management of resource management. In software development environments, the tools help to achieve continuous integration, control of program versions, automatic tests, and continuous deployment. Operations management is supported in tools for automatic deployment of applications, a configuration of virtual machines, automatic execution of scripts.

DevOps is positioned from the beginning of the iteration planning as part of the joint integration of developer and operator knowledge of the objectives of the project stages. While the developed application is in production, the DevOps team will be responsible for addressing the changes in the versions of the applications.

From the generation of the code to the deployment in production of the corresponding build, a workflow is established, and some tools and technologies allow its automation. Basically, it is a workflow that needs to be adapted according to the case, so we then make a proposal about a generic workflow and some technological reference solutions to better understand the process and its benefits.

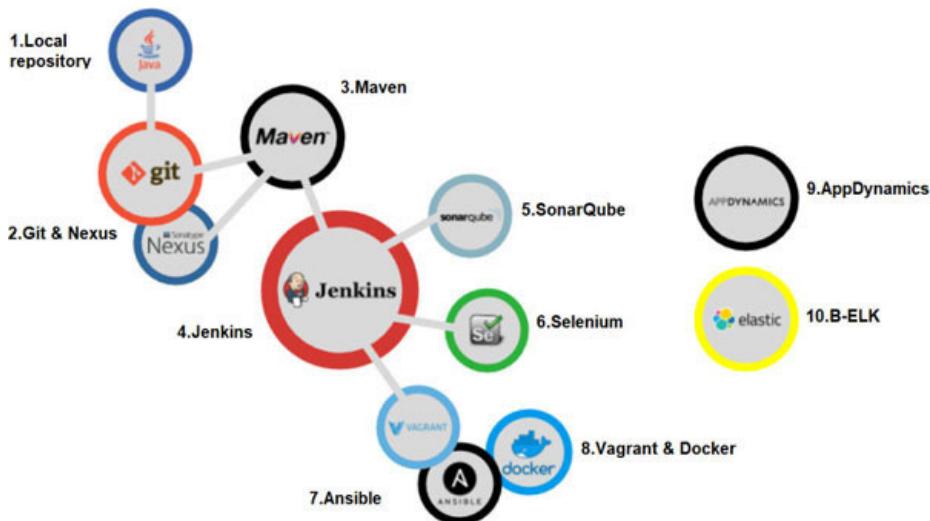


Figure 1.6: DevOps tools

Following the previous graphic, we start from the source code that a developer has on his local computer, where he works with a specific IDE (Eclipse, NetBeans, IntelliJ ...) on a specific language.

The first thing we should have is a distributed code repository. In this case, we can use Git <https://git-scm.com/> (being able to implement any of its options, hosting it in own or external hosting, such as GitLab or GitHub) so that the development team can work in a collaborative way, where all the code of the different developers is hosted in a centralized way is the first step towards continuous integration. The benefits of using this kind of tools are:

We can version the code, being able to recover a specific version in a given moment, reducing the cost and the effort of undoing changes in the code.

We will make sure that all the developers build their code on the same version and the integration problems are reduced.

Project change management is simplified, any change is tagged and traceable, and only specific changes are updated, the entire project and the changed files are not replaced.

Next, we have a dependency manager such as Maven <https://maven.apache.org/>, which is responsible for analyzing the dependencies that the project has and resolve them to compile later. From that moment, it will compile and execute the quality processes (unit tests, integration ...) that

have been established. The benefits of using this kind of tools are:

Greater control of the correct generation of the builds.

Automatic execution of the tests and verification of the results.

Once at this point, we are going to introduce Jenkins <https://jenkins.io/>, which will act as an orchestrator of multiple flow processes. You can think Jenkins as the responsibility of performing the routine tasks and check if one step of the flow to bring the functionality to production has been correct to trigger the next, which is possible thanks to the infinity of plugins that counts and that it will allow us to adapt the flow to our needs. The most common flow would be, the compilation of the project with Maven when a member of the team makes changes in the repository and the publication of the project in a production environment. It is also often integrated with Nexus repository <https://repository.apache.org> to get libraries and different versions for productive environments.

Jenkins will execute Sonar <https://www.sonarqube.org/> to verify the quality of the written code, with respect to the established metrics. If the results of the previous process are correct, Jenkins will launch the Selenium <http://www.seleniumhq.org/> process to execute the established test cases and ensure that the new code does not break the functionality collected in such cases. If this step ends successfully, we will have a build, ready to be deployed. We will then have achieved a continuous integration environment that will allow you to accelerate the process of releases generation.

Ansible <https://www.ansible.com> is a tool that allows us to manage configurations, resource provisioning and automate the deployment of the infrastructure, for example, for automating processes of CD and CI. It also allows us to install applications, orchestrate services and more advanced tasks. Ansible allows different forms of configuration. Either by means of a single file, called a playbook, which must contain all the parameters to do a specific task, on a specific group of clients; or, through a directory structure, for each project, separating the parameters into files, which can later be imported from other playbooks.

If we want to extend the concept of CI, we could include the automated deployment of the generated build and we will talk about CD. Thanks to virtualization tools such as Vagrant or Docker and configuration management tools such as Ansible (which allows making specific configurations about our virtualizations), we can be able to create the necessary environments to deploy the build.

In a complementary way, we also propose the use of an APM (application performance monitor) such as AppDynamics <https://www.appdynamics.com/>. This tool covers during the development phase, analyzing the performance of each software module developed to ensure its correct performance and that the implemented solution behaves as expected before taking it to production. Saving both time, effort and the associated cost of bringing software that does not work as expected to production can become a bottleneck of our system. In production, it allows to analyze the performance of the system and verify that in the real environment, it responds in an unexpected way.

And finally, due to the large amount of information that different parts of our system will generate, we propose the use of a logging tool such as B-ELK being a suite composed of different tools such as:

Beats <https://www.elastic.co/products/beats> : It is the solution of the suite that uses agents that are responsible for capturing the information that will later process Logstash and Elasticsearch.

Elastic search <https://www.elastic.co/products/elasticsearch> : It is the main piece of the suite, based on the search and indexing engine Apache Lucene and is responsible for storing the information.

Logstash <https://www.elastic.co/products/logstash> : It is a logs processor that allows structuring the information by formatting it and enriching it, before storing it, so that its subsequent exploitation is optimized.

Kibana <https://www.elastic.co/products/kibana> : It is a visualization tool that allows the creation of dashboards that show in a graphic way the information sent by Beats, enriched by Logstash and stored by Elasticsearch. It allows in this way to add a large amount of information for tracking KPIs and metrics, both business and IT, providing a very valuable vision for different organizations.

A good reference is this periodic table of DevOps tools made by Xebialabs (<https://xebialabs.com/periodic-table-of-devops-tools/>) that has become, thanks to its continuous updates in a guide of reference tools or, in the face of new needs, source of information to discover new ones:

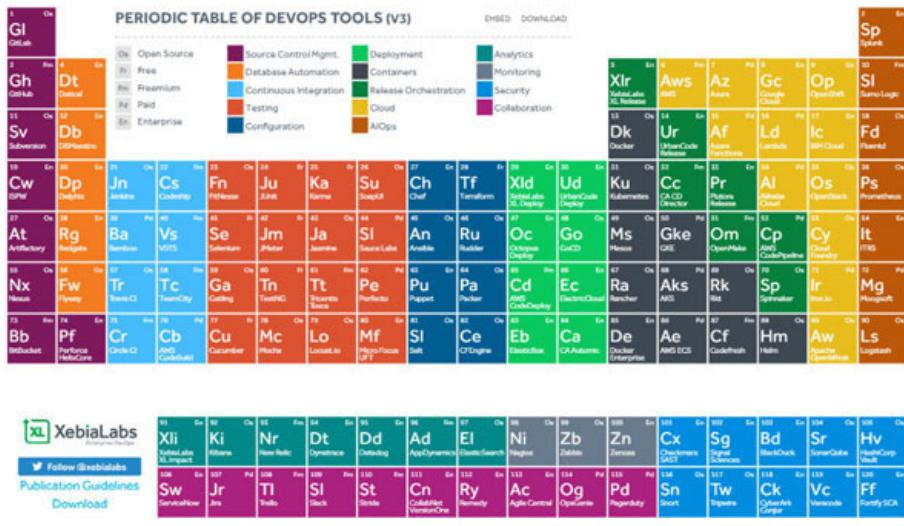


Figure 1.7: Periodic table of DevOps tools

We have different categories: security, continuous integration, cloud, monitoring, etc.. In each of them, it is also indicated if they are paid or free, and we can get more details if we click one item in particular. For example, in the case of Ansible

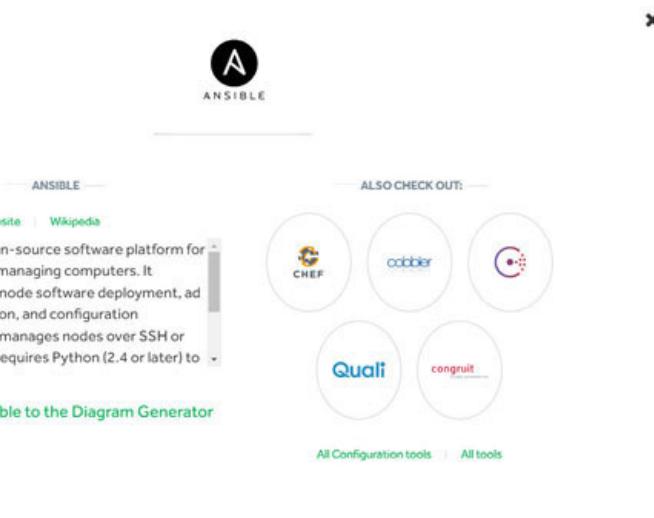


Figure 1.8: Ansible description and links

And GitHub:

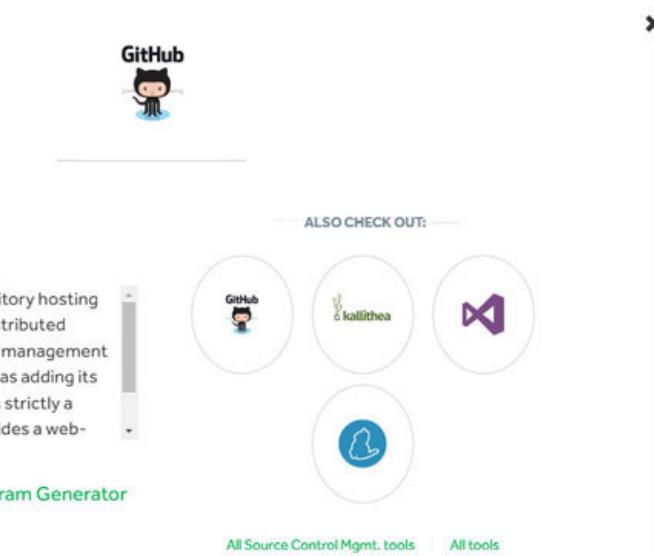


Figure 1.9: GitHub description and links

This table is an interesting point of reference for a wide variety of tools that are part of the DevOps ecosystem. It also allows us to have an overview of how all these tools are integrated into an environment, from the repository where the code is hosted, to automation in the provision of infrastructure, orchestration of services and even the monitoring of our environment. These are some of the tools organized by categories:

Continuous integration: Jenkins, Drone

Code management: GIT

Test automation: Selenium, Appium

Infrastructure management: Chef, Puppet

Repositories: Nexus

Monitoring: Graylog, Elastic Stack

Architecture: Docker, Kubernetes, OpenShift, Nomad

Cloud: AWS, Azure, Google

Automation with Jenkins and Drone

The execution of automated tasks and their orchestration is done with Jenkins. The idea is that any task that is going to be executed many times, must be configured to be executed through Jenkins since, it contributes a great value in what metainformation refers to who has executed the task, why (in case of being executed by a trigger), how (with what parameters), when, how long the execution has taken and other execution parameters.

This automation server is developed in java, is free and is open source, has hundreds of plugins to expand its functionality and integrate with different systems.

Jenkins <https://jenkins.io/> is one of the main tools of any DevOps environment for continuous integration, and it intends to carry out automatic integrations of a project as often as possible in order to detect failures in the process as soon as possible. Among the main features we can highlight:

Integration means the compilation and execution of tests of an entire project

Offers a stable compilation and packaging environment

Help prevent and solve integration problems

Strengthens the implementation of methodologies and procedures for development, QA and operations

Get programmers to devote less time to correct mistakes ... and can devote their effort to implement new functions!

Great help in the automation of tests

Generates code quality metrics

Automates deployment by environments generates special demonstration packaging

From version 2 Jenkins incorporates a fundamental feature called Jenkins Pipeline, a set of predefined plugins with the aim of allowing the configuration of pipelines of CD in a simple way, that is, to facilitate the software deployment from the code repository to different systems so that it can be used by clients and users.

Drone <https://github.com/drone> is a lightweight and undemanding continuous integration platform. The tool, written in Go, is used to load Builds (each one of the phases in the development of new software) automatically from Git repositories like GitHub, GitLab or Bitbucket and execute them in Docker containers in order to test them. The user has the possibility to run any test suite and schedule the sending of reports and status notifications to his mail. For each software test, a new container is generated based on an image of Docker's public repository, in such a way that any publicly available Docker image can be used as the environment for the code that is to be tested.

A drone is integrated into Docker and supports various programming languages such as PHP, Node.js, Ruby, Go or Python. The only dependence needed is precisely the container platform. Which means it is important to do this in a framework in which Docker is installed to build a continuous integration platform with Drone.

Drone supports various version control repositories. In the project documentation, there is a manual with the name `readme.drone.io` for the

standard installation with GitHub integration. To administer the CI platform, a web interface is used through which software builds are loaded from any Git repository, applications are grouped, and the result is executed in a previously defined test environment.

Infrastructure and configuration management

This category groups those tools designed for the simplification of automation and orchestration in the execution of environments such as scripts, recipes, blueprints, playbooks, charms and templates. In this category, we can highlight tools related to the configuration management such as Puppet, Chef, Ansible and Vagrant.

Vagrant <https://www.vagrantup.com/> is a HashiCorp tool that allows you to lift light environments, focused on development in a simple and fast way. It allows reproducing the development environment locally with the focus on reproducibility and automation. It could be said that it is a frontend since basically, it raises virtual machines or resources in different systems (VirtualBox, OpenStack, VMware, Docker, ...) through code files.

The fact that it allows to define these resources in code files instead of in proprietary files (OVA, VMDK, ...) gives vagrant a great sense within the DevOps philosophy by allowing to define as part of the infrastructure code. Vagrant is a free tool, open-source and developed in ruby.

Chef <https://www.chef.io/solutions/devops/> and Puppet <https://puppet.com/> are the benchmarks in terms of infrastructure automation, and they are configuration management tools. There are some differences between both, but in a general view, they could be compared in terms of tools that enable the codification of the infrastructure and the deployments on these defining the final state that is desired for the different servers.

A highly automated IT infrastructure is ideal for the development team to automatically deploy the code at each juncture, from the testing phase to the deployment environment. The Infrastructure as a Code (IaaS) is a potent antidote for the possible bottlenecks of the Dev and Ops teams. An infrastructure of this type should include infrastructure provisioning tools, which create and deploy infrastructure by clicking or quickly completing a template. It should also include configuration management tools that facilitate, for example, the upgrade of many servers with a single command.

Very similar to the previous ones is Ansible <https://www.ansible.com/>, but with the possibility of orchestrating the deployments from the controller itself, that is, the node that executes the ansible code (accessing the nodes by SSH), something that the previous tools can't do in case alone and for what could be considered ansible as an orchestration tool, although the usual use of ansible is to use it as a configuration management tool.

These tools allow both an analyst of the systems team, or infrastructure and networks, or a programmer to deploy the necessary components in a matter of minutes, either to do code tests, for a demonstration or to check collateral effects of a patch. Also, allow the automation of critical and repetitive tasks associated with the management of the systems infrastructure:

Install the operating system on a new computer or perform an update on an existing computer

Update an application or library

Install a new service (Apache, IIS, Tomcat, WordPress, etc.) or change the configuration of an existing service

Update a database engine or update its users

Generate SSH keys on a server

Update SSL certificates

Monitoring tools

The monitoring must be something transversal to all development teams with the target for getting feedback about the results of developments. In this category, we can highlight application focused to performance such as Zabbix, Nagios, Influxdb, Telegraf, and Grafana, log revision such as GrayLog, Logstash, Kibana, and Elasticsearch, web analytics with Google Analytics.

Nagios and Zabbix are the classic monitoring tools that have evolved adapting to the philosophy of DevOps work teams, competing for both to become the de facto standard for infrastructure monitoring. Both are free and open-source and written mainly in C.

Prometheus is taking some relevance, together with Grafana, allows the visualization of metrics stored in its database as time series. It is free, open-source and is written in Go.

ELK stack and Kibana

Elasticsearch, Logstash and Kibana, also named "ELK stack", are three tools of great potential. United can offer a powerful solution in the detection of incidents in an IT organizational structure. Elasticsearch plays the role of a search server where data already optimized by indexing is stored. Logstash is the parser of the data that comes from different sources and that, filtered, homogenize the message, and you can get to dispense with the part of the message that is not important. Kibana is the front-end for the visualization and analysis of data. Each of them can be used as an independent tool, but the union of all of them makes a perfect combination for records management.

Kibana is a free distribution platform to perform data analytics, mainly logs of infrastructure servers that are stored in the Elasticsearch database. In a simple way, you can analyze the data and make graphs, tables and maps that will facilitate the visualization of the data.

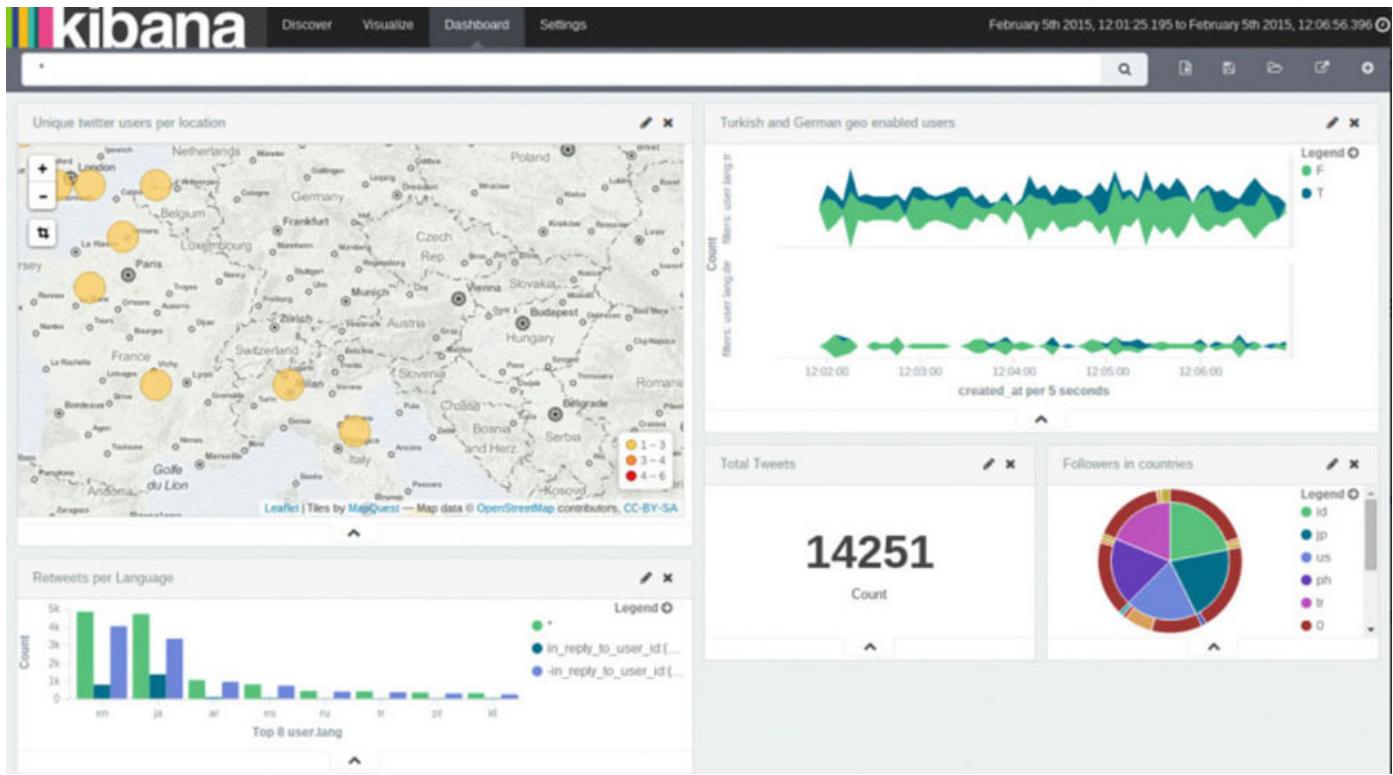


Figure 1.10: Kibana as a platform to perform data analytics

Kibana makes it easier to understand a large volume of data. From an interface based on the browser (Mozilla Firefox, Chrome, IE, Safari, ...) it allows to quickly access the data and create dynamically dashboards that can contain different queries to Elasticsearch in real-time.

Containers and orchestration

The containers have changed the way in which the software is packaged, distributed and deployed within all the diversity of technologies that allow this light virtualization. Docker, it is the best known and is being widely accepted by the industry against other options such as LXC or rkt. It is free, open-source and is written in Go.

Related to container technology is Kubernetes, developed by Google for its own use, it is a container orchestrator, allowing more convenient management of these, allowing things like autoscaling. Like docker is written in Go and is free and open-source (it was donated by Google to the Cloud Native Computing Foundation, part of the Linux Foundation).

Other container orchestration options are OpenShift from RedHat, and basically, it is a Kubernetes system covered with numerous plugins or additional management systems, Nomad from Hashicorp, Rancher or Docker's own Swarm.

All these orchestrators allow the definition of the services through files with different formats, which facilitates the distribution of the deployments. A solution to define deployments of services but that does not require as much infrastructure as to use an orchestrator is to use docker-compose <https://docs.docker.com/compose/>, which allows the definition of services with multiple containers, networks, etc. through configuration files in YAML format.

DevOps and security

DevOps is not just about the development teams and operations. In order to take full advantage of the DevOps approach, companies must consider the role that security plays in the lifecycle of their applications. This means thinking about basic security from the planning stage onwards.

It also involves automating some security features to prevent the DevOps workflow from slowing down. Selecting the right tools to integrate security can be useful to achieve your DevOps security goals.

But the effective security of DevOps requires more than new tools; it is built on the cultural changes of DevOps to integrate the work of the security teams as soon as possible. DevOps streamlines everything by shortening the distances between development and operations, but the agility obtained can be impaired by insufficient security planning.

Security used to be the sole responsibility of an isolated team and was added at the final stage of development. Currently, in a collaborative DevOps framework, security is a shared and integrated responsibility from the start.

An introduction to DevSecOps

Currently, in the collaborative work framework of DevOps, security is a shared responsibility and integrated throughout the process. It is such an important approach that it led some to coin the term DevSecOps to emphasize the need to create a security base in the DevOps initiatives.

DevSecOps means thinking from the beginning about the security of applications and infrastructure. It also involves automating some security doors to prevent the DevOps workflow from slowing down. Selecting the right tools to y integrates security continuously can help you achieve your security goals, but the effective security of DevOps requires more than new tools; it is built on the cultural changes of DevOps to integrate the work of the security teams as soon as possible.

In part, DevSecOps highlights the need to invite security teams from the start of DevOps initiatives to develop information security and establish a plan for security automation. It also underlines the need to help developers codify with security in mind, that is, a process that involves security teams sharing visibility, comments and information about known threats. This may also include new security training for developers, as more traditional application development has not always emphasized security.

The adoption of DevOps practices means more collaboration between development, QA, IT, and operation teams, and more in-progress adoption of continuous integration or continuous delivery tools. This provides a good foundation to move to DevSecOps. The diagram below shows the security involved with the development, QA, and operations through the whole CI/CD lifecycle.

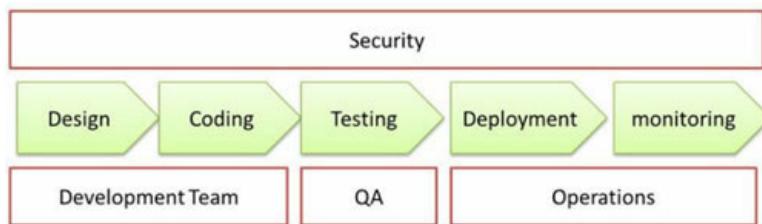


Figure 1.11: DevSecOpsLifeCycle

It is recommended to maintain short and frequent development cycles, integrate safety measures with minimum disruption of operations, keep up with innovative technologies such as containers and microservices. All these initiatives start at the human level, with the details of collaboration in your organization, but the facilitator of those human changes in a DevSecOps framework is automation.

But what should be automated and how to do it? There is a written guide <https://itrevolution.com/book/devops-and-audit/> that will help answer this question. Organizations must take a step back and consider the entire development and operations environment. This includes source code control repositories, container registers, continuous integration and continuous implementation process (CI / CD), application programming interface (API) management, automation of coordination and releases, and operational management and supervision.

New automation technologies have helped organizations to adopt more agile development practices and have also participated in the advancement of new security measures. But automation is not the only thing in the IT landscape that has changed in recent years: native cloud technologies, such as containers and microservices, are now an important part of most DevOps initiatives, and security of DevOps must adapt to meet them.

The larger scale and more dynamic infrastructure enabled by the containers have changed the way many organizations do business. Because of this, DevOps security practices must be adapted to the new landscape and adjusted according to container-specific security guidelines. Native cloud technologies do not lend themselves to security policies and static checklists. Rather, security must be continuous and must be integrated into each stage of the application's lifecycle and infrastructure.

DevSecOps means integrating security into the development of applications from start to finish. This integration into the process requires a new organizational approach that is also necessary for the new tools. With that in mind, DevOps teams should automate security to protect the environment and data in general, as well as the continuous integration / continuous delivery process, a goal that will likely include microservice security in containers.

Conclusion

An analysis of the DevOps principles has been carried out as continuous integration, continuous delivery, continuous deployment, which allows visualizing the culture as a collaborative paradigm in the software development, and that together with agile methodologies produce effective results in short and orderly times.

It has studied the components in which the borders and roles DevOps are developed with their activities: Culture, Sharing, Automation and Metrics, which allow giving a clear idea of the good practices that must be structured to carry out the development and software deployment continuously effectively.

As a summary, implementing DevOps in a company allows:

Improve collaboration in development, gaining control and traceability over the code that is generated. Reducing the risk during code merges and accelerating the integration process.

Accelerate the process of generation of builds and improve the quality of the code that is incorporated in these builds, advancing problems and reducing risks and impact derived from putting a code that does not work correctly in production.

Automate repetitive manual processes that do not add value, reducing the publication cycle and significantly improving the time-to-market of the versions and their promotion between environments.

Automate the horizontal scaling of a system so that once a capacity planning is reviewed, it is possible to resize the platform easily.

All the benefits are aimed at improving the quality of the software, reducing the delivery time of functionality and saving efforts and costs in the entire process, aligning the development and production strategies.

The success of DevOps depends to a large extent on the intensive and coordinated collaboration between the client, the developer and those responsible for IT operations. Developers should focus on coding and IT operations teams in the automated infrastructure management. Both need to talk to each other to discover new ways to innovate and improve both processes and deliverables. Working in silos without communication is a thing of the past.

CHAPTER 2

Container Platforms

In this chapter, we will review the main containers platforms that provide common tooling for both development and operations teams and the ability to use rapid development processes with managed release processes. Containers platforms such as OpenShift combines Docker containers with Kubernetes orchestration project and includes centralized administration, such as for instance management, monitoring, traffic management, and automation.

Structure

Docker containers

Container orchestration

Kubernetes

Docker swarm

OpenShift container platform

Objectives

Understanding the concept of Docker containers

Understanding container orchestration

Knowing about container platforms like Kubernetes, Docker swarm and OpenShift

Docker containers

The aim of the DevOps is to improve the quality of the new software versions and to accelerate the development, delivery, and implementation thanks to much more effective cooperation of all those involved and to the continued automation. Automated DevOps tasks include repository build processes, static and dynamic code analysis, and module, integration, system, and performance testing. The core spine of DevOps is still the reflections on continuous integration (Continuous Integration, CI) and continuous delivery (Continuous Delivery, CD), two central fields of application of the Docker platform.

What is Docker?

Docker offers integration options for consolidated CI/CD tools such as Jenkins, Travis, or Drone and allows you to load the code automatically from

the Docker Hub or version control repositories such as GitHub, GitLab or Bitbucket. This is how the container platform represents a base for DevOps workflows in which developers can create new components for applications in common and run them in any testing environment.

Docker is a container platform to develop, deploy and manage applications quickly. Docker packages software into standardized units called containers that include everything necessary for the software to run, including libraries, system tools, and code. With Docker, you can deploy and adjust the scale of applications quickly in any environment with the certainty of knowing that your code will run the same, from the development to the production environment and both in the cloud and on-premise.

Docker uses the LibContainer to manage the functions of the Linux Kernel and uses a group of isolation technologies such as Namespaces, Control Groups, AppArmor, security profiles, network interfaces, and rules for the firewall necessary for the operation of the containers.

A remarkable feature of this container is the Docker Hub <http://hub.docker.com/>, a repository where Docker users can share the images they have created with other users. For Linux users, installing one of these containers is as easy as downloading an application from the App Store. The download from the Docker Hub is done through commands and runs on the system itself.

Docker new features for container management

Docker implements a high-level API to provide lightweight virtualization, that is, lightweight containers that execute processes in isolation. This is achieved mainly by using two features of the Linux kernel: Cgroups and namespaces, which provide us with the possibility of using resource isolation (CPU, memory, I/O block, network, etc.).

With the use of containers, resources can be isolated, services are restricted, and processes are given the ability to have an almost completely private vision of the operating system with its own process space identifier, the structure of the file system, and the network interfaces. Multiple containers share the same core, but each container can be restricted to using only a defined amount of resources such as CPU, memory, and I/O. Some Docker features are:

It is light since there is no complete virtualization, taking better advantage of the hardware and only needing the minimum file system for the services to work.

The containers are self-sufficient (although they can depend on other containers), not needing anything more than the image of the container for the services offered to work.

A Docker image could be understood as an operating system with installed applications. A container can be created from an image. The docker images are portable between different platforms, and the only requirement is that docker is available in the host system.

The project offers us a repository of images in the GitHub style. This service is called Registry Docker Hub and allows you to create, share, and use images created by third parties or by us.

Virtualizing with Docker offers us a series of advantages over doing it with conventional virtual machines:

Portability: All containers are portable, so we can easily take them to any other Docker device without having to reconfigure anything. Docker allows you to run your application locally on any operating system, on an on-premise server, or even in the cloud.

Lightness: By not virtualizing a complete system, but only what is necessary, the consumption of resources is minimal. The saving of resources is around 80%.

Self-sufficiency: Docker is responsible for everything, so the containers should only have what is necessary for the application to work, for example, those libraries, files, and configurations necessary to perform its function.

Performance: Containers have better performance than traditional virtualization since it is based on LXC (Linux Containers), which runs directly on the kernel of the host machine, avoiding the traditional virtualization layer based on a hypervisor that penalizes performance.

Docker architecture

A Docker container system consists mainly of 5 elements:

Docker Engine (daemon): It is a daemon that runs on any Linux distribution, and that exposes an external API for the management of images and containers. With it, we can create images, upload them, and download them from a docker registry and execute and manage containers.

Docker client: The Docker client allows us to manage the Docker engine and can be configured to work with a local or remote Docker engine, allowing us to manage both our local development environment and our production environment.

Docker image: Template used to create the container for the application that we want to deploy.

Docker registry: Directories where the images are stored, both public and private access. The purpose of this component is to store the images

generated by the Docker engine. It can be installed on a separate server and is a fundamental component since it allows us to distribute our applications.

Docker containers: Folders where everything necessary (libraries, dependencies, binaries, etc.) is stored so that the application can run.

Figure 2.1: Docker architecture

Docker engine

The heart of any Docker project is the Docker engine, that is, an open-source client-server application available to all users in the current version on all established platforms. The components that make up the basic architecture of this engine are a daemon with server functions, a programming interface (API) based on REST (Representational State Transfer), and the terminal of the operating system (Command-Line Interface, CLI) as an interface of the user (client).

Docker daemon: Docker engine uses a daemon process as a server that works in the background of the host system and allows central control of the Docker engine. It is also responsible for creating and managing all images, containers, or networks.

The API REST: Specifies a series of interfaces that allows other programs to interact with the daemon and give instructions. One of these programs is the terminal of the operating system.

The terminal: Docker uses the terminal of the operating system as a client program, which interacts with the daemon through the REST API and allows users to control it through scripts or commands.

Docker allows executing, stopping, or managing software containers directly from the terminal. With the docker command and instructions like build (create), pull (download), or run (execute), it is possible to communicate with the daemon, which makes it possible for both client and server to be in the same system. In addition, it is possible to address the daemon in another different system. Depending on the type of connection to be established, communication between client and server occurs either through the REST API, UNIX socket, or a network interface.

The docker run command starts the Docker daemon to search for and start a container with the name hello-world. If Docker has been installed correctly, you should receive an output like the one shown in the following image:

Figure 2.2: Executing Docker run command

After downloading the image successfully and receiving the message Downloaded newer image for hello-world: latest ("the latest hello-world: latest image has been downloaded"), the container is started, which includes a simple hello-world script. You can share images, automate workflows, and more with a free Docker ID: <https://cloud.docker.com>. For more examples visit: <https://docs.docker.com/engine/userguide>

Docker registry

The Docker registry is an open-source project that can be installed free on any server, but Docker offers Docker Hub <http://hub.docker.com>, a paid SaaS system where you can upload your own images, access public images of other users, and even official images of the main applications such as MySQL, MongoDB, RabbitMQ, Redis, etc.

Docker client

The Docker client makes use of the remote API of the Docker engine and can be configured to talk with a local or remote Docker engine, allowing us to manage both our local development environment and our production servers. The most common docker commands are:

docker info: Gives information about the number of containers and images that the current machine is managing, as well as the plugins currently installed.

docker images: List information of the images that are available on the machine (name, id, space it occupies, the time that elapsed since it was created).

docker build: Create an image from the Docker file of the current directory.

docker pull : Download the indicated image version to the current machine. In case of not indicating the download version, all that is available.

docker push : Uploads the version of the indicated image to a Docker registry, allowing its distribution to other machines.

docker rmi: Deletes an image of the current machine.

docker run : Create a container from an image. This command allows a multitude of parameters, which are updated for each version of the Docker engine, so for its documentation, it is best to refer to the official page.

docker ps: Shows the containers that are running on the machine. With the flag -t it also shows the containers that are stopped.

docker inspect container: Shows detailed information of a container in json format. You can access a particular field with the command `docker inspect -f '{{.Name}}' container`

docker stop : For the execution of a container.

docker start : Resumes the execution of a container.

docker rm : Delete a container. To delete all the containers of a machine, you can execute the command `docker rm -fv $(docker ps -aq)`

docker logs : Shows the logs of a container.

docker stats : Shows the execution statistics of a container, such as the memory used, the CPU, the disk, etc.

docker exec : Executes a command in a container. Useful to debug containers in execution with the options `docker exec -it container bash`.

docker volume ls: Lists the existing volumes on the machine. For a complete list of commands related to volumes run `docker volume --help`

docker network ls: Lists the existing networks on the machine. For a complete list of commands related to networks, run the docker network --help.

docker exec: Execute a command in a container. Useful to debug containers in execution with the options docker exec -it container bash

docker cp: Copy files between the host and a container.

docker logs: Shows the logs of a container.

docker stats: Shows the execution statistics of a container.

The docker command line will connect to this daemon, which will keep the docker status and so on. Each of the commands will also be executed as a superuser, by having to contact this daemon using a protected socket. From there, we can create a container by downloading it from the official repository.

```
$ docker pull ubuntu
```

The pull command downloads a basic Ubuntu container and installs it. There are many images created and can be created and shared on the Docker website, in the style of Python libraries or Debian packages. You can search all the images of a certain type, like Ubuntu, or look for the most popular images.

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
5b7339215d1d: Pull complete
1aca88e9f672: Pull complete
a31c3b1caad4: Pull complete
b054a26005b7: Pull complete
Digest: sha256:9b1702dcfe32c873a770a32cf306dd7fc1c4fd134adfb783db68defc8894b3c
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
[node1] (local) root@192.168.0.23 ~
$ docker images
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
ubuntu              latest         4c108a37151f      3 weeks ago        64.2MB
```

Figure 2.3: Executing docker pull command

Once downloaded, you can start to execute commands. The good thing about docker is that it allows you to execute them directly without having to connect to the machine. We can execute, for example, a shell when running the container:

```
$ docker run -i -t ubuntu /bin/bash
```

This indicates that a terminal(-t) is being created, and the command is being executed interactively (-i). In the previous instruction, we are executing the command /bin/ bash for getting a terminal shell inside the container.

```
$ docker run -i -t ubuntu /bin/bash
root@760dca7304c6:/# ls -l
total 4
drwxr-xr-x    2 root root 4096 Jun 12 16:55 bin
drwxr-xr-x    2 root root     6 Apr 24 2018 boot
drwxr-xr-x    5 root root  360 Jul 16 09:10 dev
drwxr-xr-x    1 root root    66 Jul 16 09:10 etc
drwxr-xr-x    2 root root    6 Apr 24 2018 home
drwxr-xr-x    8 root root   96 May 23 2017 lib
drwxr-xr-x    2 root root  34 Jun 12 16:55 lib64
drwxr-xr-x    2 root root    6 Jun 12 16:54 media
drwxr-xr-x    2 root root    6 Jun 12 16:54 mnt
drwxr-xr-x    2 root root    6 Jun 12 16:54 opt
dr-xr-xr-x  1179 root root    0 Jul 16 09:10 proc
drwx-----    2 root root  37 Jun 12 16:55 root
drwxr-xr-x    1 root root   21 Jun 18 22:51 run
drwxr-xr-x    1 root root   21 Jun 18 22:51 sbin
drwxr-xr-x    2 root root    6 Jun 12 16:54 srv
dr-xr-xr-x    13 root root    0 Jul 15 01:26 sys
drwxrwxrwt    2 root root    6 Jun 12 16:55 tmp
drwxr-xr-x    1 root root   18 Jun 12 16:54 usr
drwxr-xr-x    1 root root   17 Jun 12 16:55 var
```

Figure 2.4: Inside the container

Testing Docker in the cloud

In the URL <https://labs.play-with-docker.com> we have a service that allows you to run Docker containers in the cloud.



Figure 2.5: Running Docker in the cloud

Play with Docker is an online environment that allows you to run Docker commands without having a docker installed on your machine. Give the experience of having an Alpine Linux virtual machine in the browser, where you can build and run Docker containers and even create clusters in the Docker Swarm mode. <http://training.play-with-docker.com>

The screenshot shows the 'Play with Docker classroom' interface. It features a sidebar with navigation links like 'First Alpine Linux Containers', 'About', 'Help', 'Logout', and 'Sign up'. The main content area displays a terminal session on a 'node1' instance.

First Alpine Linux Containers

Sep 19, 2017 • @jimmodified

In this lab you will run a popular, free, lightweight container and explore the basics of how containers work, how the Docker Engine executes and isolates containers from each other. If you already have experience running containers and basic Docker commands you can probably skip this intro exercise.

Concepts in this exercise:

- Docker engine
- Containers & images
- Image registries and Docker Store (AKA Docker Hub)
- Container isolation

Tips:

Code snippets are shown in one of three ways throughout this environment:

- Code that looks like **this** is sample code snippets that is usually part of an explanation.
- Code that appears in box like the one below can be clicked on and it will automatically be

If the commandline doesn't appear in the terminal, make sure popups are enabled or try resizing browser window.

```
[node1] (local) root@192.168.0.43 ~
$ #####
#           WARNING!!!!
# This is a sandbox environment. Using personal credentials
# is HIGHLY! discouraged. Any consequences of doing so are
# completely the user's responsibilites.
#
# The PWD team.
#####
$ docker info
Client:
  Debug Mode: false
Server:
  Containers: 0
    Running: 0
    Paused: 0
    Stopped: 0
  Images: 0
  Server Version: 19.03.0-beta2
```

Figure 2.6: Play with Docker

Play with Docker also includes a training site composed of a set of laboratories with practices from basic to advanced levels.

Container orchestration

The rise of containers has changed the way programmers conceive the development, deployment, and maintenance of software applications. Using the native isolation capabilities of modern operating systems, containers support a separation of interests (Separation of Concerns, SoC) similar to that of virtual machines, but without consuming as many resources and with greater flexibility of deployment in comparison with virtual machines based on hypervisors.

The containers are so light and flexible that they have given rise to new architectures of applications. This new approach consists of packaging the different services that constitute an application into separate containers and then deploying those containers through a cluster of physical or virtual machines.

But nowadays, the applications are complex, and as a rule, it does not come with deploying a single container in production, except in the simplest cases. The usual thing is to need several, which must also scale differently and other complexities. For example, one container for the front-end, one or more for the service interface and another for the database

All this gives rise to the need for container orchestration, that is, having a tool or system that automates the deployment, management, scaling, interconnection, and availability of our container-based applications. A container orchestrator is responsible for the following tasks:

Deployment and raised automatic container-based services.

Self-scaling and load balancing.

Control of the health of each container.

Maintenance of secret parameters and configurations.

Docker compose

Docker-compose allows you to connect several containers and execute them with a single command. Implemented in the Python scripting language, its fundamental component is a central control file based on the YAML markup language. The syntax of this file resembles that of open source software Vagrant, used in the creation and provisioning of virtual machines.

Docker-compose allows you to define a series of containers and the relationships between them at the level of YML file with a very intuitive format. Given this YML file, it is responsible for orchestrating the creation of the containers in the correct order. It is also capable of detecting the definitions that have changed from one YML file to another, and relaunch only those services that have changed.

In the file docker-compose.yml , you can define as many software containers as you want, including all the dependencies as well as their interrelationships. The scheme followed to manage the multi-container applications does not differ from that needed to manage simple containers. With the docker-compose command, the corresponding subcommand manages the entire life cycle of the application.

Another characteristic of Docker compose is its integrated scaling mechanism, through the command line program, with this Docker tool, you can define how many containers are to be started for a given service.

You can read the official documentation <https://docs.docker.com/compose/gettingstarted/> for more information and examples.

Kubernetes

Kubernetes <https://kubernetes.io> , also known as K8S, is the most popular container orchestration engine on the market and is an open-source orchestrator for applications that run in software containers, automating the deployment, scalability, and management of distributed applications.

The reception of Kubernetes was so great that the project was adopted by the community at the head of the CNCF - Cloud Native Computing Foundation <https://www.cncf.io> , an organization created as part of the Linux Foundation. The open-source Kubernetes, currently in a fairly high state of maturity, are managed by the CNCF. With the foundation, the project is no longer headed by a single company and is developed with the support (and interests) of dozens of organizations and thousands of members of the community.

Kubernetes groups the containers in logical fragments called pods, which represent the basic units of the manager, which can be distributed in the cluster by the scheduling method. Kubernetes is based on the master-slave architecture: a cluster consists of a master (Kubernetes master) and other slaves or Kubernetes nodes (also called workers). The master acts as a central control level (control plane) in the cluster and is composed of four basic elements that allow coordination within the cluster and distribute tasks: an API server, configuration memory, etc., a scheduler and a controller manager.

API server: In a Kubernetes cluster, all automation is launched in an API server by means of a REST API. This server acts as the central management point in the cluster.

etcd: It's an open-source key-value store and can be considered the memory of a Kubernetes cluster. Developed by CoreOs, especially for distributed systems, etcd, stores configuration data and facilitates it to the nodes.

Scheduler: The role of the scheduler is to distribute the pods in the cluster, for which it finds out how many resources a Pod needs and adjusts them with the resources available to each node in the cluster.

Controller manager: this is a service of the Kubernetes master that regulates the status of the cluster and executes routine tasks, thus directing the orchestration. The main obligation of the controller manager is to ensure that the state of the cluster corresponds to the state that was previously defined as objective.

While the master is responsible for the orchestration, the distributed pods in the cluster are run on different nodes called workers. To do this, each node has to run a container engine, Docker, in practice, although Kubernetes is not linked to any specific container engine. In addition to the container engine, the Kubernetes nodes also include these components:

kubelet: With this name, an agent is designated who, running in each node, directs and manages it. As the main point of contact in the nodes, the kubelet maintains the communication and ensures that the information is sent to the control level and that it is received. The agent receives the requests and supervises their execution in each of the nodes.

kube-proxy: Next to the container engine and the kubelet agent, in each node of Kubernetes also runs this proxy service in charge of requests that arrive from abroad to be sent to the corresponding container and to provide services to the users of containerized applications.

While Docker handles entities that are referred to like images and containers, Kubernetes wraps those entities in what they refer to as pods. A pod can contain one or more containers running and is the unit that Kubernetes manages. There are several advantages that Kubernetes brings to the administration of containers as pods:

Multiple nodes: Instead of simply deploying containers on a single host, Kubernetes can implement a set of pods on multiple nodes. Essentially, a node provides the environment where a container runs.

Replication: Kubernetes can act as a replication driver for a pod. This means that you can set how many replicas for a specific pod.

Services: The word service in the Kubernetes context implies that you can assign a service name (ID) to a specific IP address and port, and then assign a pod to provide that service. Kubernetes internally tracks the location of that service to redirect requests for another pod from that service to the correct address and port.

Kubernetes installation & key terms

If you want to install Kubernetes in your local machine, you can use minikube <https://github.com/kubernetes/minikube> .

Also, we can install and deploy a Kubernetes cluster with kubeadm <https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm> .

These are some terms that we should understand when we go deeper into Kubernetes:

Cluster: These are physical or virtual resources and storage resources used by Kubernetes, where the pods are deployed, managed, and replicated. Kubernetes can be used in different systems such as Debian, Ubuntu, RedHat, among others.

Pods <https://kubernetes.io/docs/concepts/workloads/pods/pod> : They are the smallest unit that includes one or more Docker containers that work under the same unit. In many cases, a pod is composed of a single container, but its ability to accommodate several containers very close to each other is a very powerful feature of Kubernetes. A pod represents a set of containers that share storage and a single IP.

Replication controllers: A replication controller is a Kubernetes mechanism that ensures that a pod has raised a certain number of replicas. If we need more replicas, the replication controller raises more replicas; if we need less, it kills them; if any of them fails and dies, then it raises new replicas to keep the number defined.

Services: Services allow access to containers with a unique DNS name and stable IP addresses. Define how to access a group of pods. In this URL <https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services---service-types> you can see the types of services you can publish in Kubernetes.

Labels: They are fundamental and are used to organize and select a group of objects in pairs of type key: value. They help get lists of the servers where the traffic should go.

If you choose to configure Kubernetes, it is important to understand the following concepts before you begin:

Kubernetes driver: A Kubernetes controller acts as a node from which pods, replication controllers, services, and other components of a Kubernetes environment are deployed and managed. To create a Kubernetes driver, you must configure and run the systemd, kube-api-server, kube-controller-manager, and kube-scheduler services.

Kubernetes nodes: A Kubernetes node provides the environment in which containers are executed. To run as a Kubernetes node, it must be configured to run the docker, kube-proxy , and kubelet services. These services must be executed on each node of the Kubernetes cluster.

Kubectl command: most Kubernetes administration is done on the master node using the kubectl command. With kubectl , it is possible to create, obtain, describe, or eliminate any of the resources that Kubernetes manages (pods, replication controllers, services, etc.).

Resource files (YAML or JSON): when you create a pod, a replication controller, a service, or another resource in Kubernetes, the kubectl command expects that the information needed to create that resource is in one of these two types of formats.

Kubernetes cloud solutions

Kubernetes is currently open-source and is used as the basis for the majority of container orchestration services. If we want to have all the advantages of Kubernetes, without having to manage everything below, we have all these alternatives in the cloud:

Google Kubernetes engine <https://cloud.google.com/kubernetes-engine/> : service managed and offered by Google, which is responsible for managing the instances of the compute engine below. It also deals with monitoring, logging, the health of the instances, and updating Kubernetes to the latest available version.

Amazon, despite having its own container orchestration system called Amazon ECS, already offers a managed Kubernetes service that it has called EKS <https://aws.amazon.com/es/eks/>

Azure has its own service based on Kubernetes which it has called AKS <https://azure.microsoft.com/es-es/services/kubernetes-service/>

IBM also offers in its cloud a managed Kubernetes service called IBM Cloud Kubernetes Service <https://www.ibm.com/cloud/container-service>

CoreOS Tectonic <https://coreos.com/tectonic> is the product through which CoreOS provides Kubernetes. It facilitates portability among several providers of public and private cloud. Its installation, updating, and maintenance require fewer operations work. This tool includes Prometheus <https://prometheus.io/> for alert monitoring and management.

Kops <https://github.com/kubernetes/kops> : used to create and manage Kubernetes clusters (if required, in production and with high availability) from the command line. So far, it has been the unofficial way to install Kubernetes on AWS, and they have plans also to include Google Compute Engine and VMware vSphere.

Mesosphere <https://mesosphere.com/blog/kubernetes-dcos> : they will lean heavily on the use of Kubernetes as an orchestrator instead of a Marathon.

CloudFoundry <https://cloudfoundry.org/container-runtime/> offers Kubernetes in its container runtime.

OpenShift <https://www.openshift.com> : the leader of the PaaS integrates Kubernetes and, when using it in its different options (enterprise, online, etc.), we will be using managed K8S clusters.

Kubernetes can be completed with a large number of tools and extensions that extend the features of the platform. Prometheus <https://prometheus.io/>, Sysdig <http://www.sysdig.org/>, and the Helm package manager <https://helm.sh/> are among the best known.

Docker swarm

Swarm <https://docs.docker.com/engine/> swarm is the solution proposed by Docker to the problems of developers when it comes to orchestrate and plan containers through many servers. Swarm comes bundled with the Docker engine from version 1.12.0 and offers many advanced integrated features such as service discovery, load balancing, scaling, and security.

Docker Swarm is an open-source native clustering tool for Docker. It converts a pool of Docker hosts into a single, virtual Docker host. As Docker swarm serves the standard Docker API, any tool communicating with the Docker daemon can use Swarm to scale to multiple hosts.

Swarm follows Docker's philosophy of focusing on the simplicity and experience of the developer. You could say it's easier to use than other solutions like Kubernetes, but not as powerful and not so adopted by companies, cloud providers, or the community.

The main elements of the swarm architecture are:

Swarm master: Responsible for the entire cluster and manages the resources of several hosts Docker.

Swarm nodes: Each node of the cluster must be accessible by the Master. Each node executes an agent so that it registers the Docker daemon referenced, monitors, and updates the backend with the state of the node.

Swarm discovery: By default, Swarm uses a discovery service, based on Docker Hub, using a token to discover the nodes that are part of a cluster. It also supports other discovery services such as etcd, Consul, and Zookeeper.

Swarm strategy: Swarm has multiple strategies for the classification of nodes. When a new container is executed, the swarm decides to place it in the node with the highest-ranking calculated for its chosen strategy.

Swarm networking: It is fully compatible with the new network model (overlay) of Docker.

Figure 2.7: Docker swarm architecture

Swarm is a native cluster solution provided by Docker. With swarm, it is possible to manage a group of Docker host resources and schedule the containers to run transparently, automatically managing the workload. To do this is constantly checking the requirements at the resource level of the container, examining the available resources, and optimizing the location of the workloads.

A cluster orchestrated by Docker swarm is a group of nodes that runs Docker. One of the nodes in the cluster acts as an administrator of the other nodes and includes containers for the programmer and the service discovery component (service discovery).

Swarm provides high availability and continuously monitors the status of containers and, if any of them suffer an interruption, automatically has the capacity to balance the cluster, moving and restarting the containers to keep it running.

The native swarm clustering tool groups a series of Docker hosts in a single virtual host and uses the Docker API so that any Docker tool that has contact with the Docker daemon can access Swarm and can be scaled in a certain number of times. hosts. Users use the CLI of the Docker engine to create swarms, distribute applications in the cluster, and direct swarm behavior. It does not require additional orchestration software.

Those Docker engines that have been joined in clusters work in swarm mode. This mode is activated to create a new cluster or add a host that already exists to a swarm. Each of the hosts in a cluster becomes a node, and these nodes can be run as virtual hosts on the same local system, although the most common variant consists of a cloud-based structure in which the nodes of the swarm are distributed in various systems and infrastructures.

At the base of this software is a master-slave architecture: when tasks need to be distributed in the swarm, users transfer a so-called service to the manager node, which acts as a master node in the cluster. The master node is responsible for planning the containers in the cluster and acts as the primary interface when accessing swarm resources.

Each Docker cluster consists of at least one master node (also called administrator or manager) and as many slave nodes (called work or workers) as necessary. While the swarm master is responsible for managing the cluster and delegating tasks, the slave is responsible for executing the units of work (tasks or tasks). In addition, container applications are distributed in services in the selected Docker accounts.

Swarm in practice

In this lab <https://training.play-with-docker.com/swarm-stack-intro>, you can introduce a docker swarm creating a cluster with 2 nodes. Now we are going to resume some of the steps.

For creating a cluster with swarm mode, we have to start from a node destined to be a manager. We can execute the following command for initializing the swarm cluster mode:

```
[node1] (local) root@192.168.0.37 ~  
$ docker swarm init --advertise-addr $(hostname -i)
```

Swarm initialized: current node (vz79cf4danb0njb3mg9p8vwpu) is now a manager.

At this point we have initialized our swarm cluster, next we will review how to add a worker to this swarm.

When executing the command, we have initialized this node as a manager. With the mandatory parameter --advertise-addr we indicate the IP of the Manager that will be used internally for Swarm connections. If we omit the port, it will take 2377 by default. The output of the command shows us two tokens. Each of them serves to join additional manager and worker nodes. We will now add a worker node to the cluster. For this and from the worker console, we execute:

```
$ docker swarm join --token SWMTKN-1-1u4oi5o8db4831ipn37cyof8attas3h0xt2bf2y0fdf2my0id5-6xje7khfhupyatgr2ifkb0u1r 192.168.0.37:2377
```

This node joined a swarm as a worker.

Now we have 2 nodes, one with IP address 192.168.0.37 and another with IP address 192.168.0.38 . The first one acts as node manager and the second one as node worker.

```
$ docker node ls
ID          HOSTNAME   STATUS    AVAILA
BILITY      MANAGER STATUS  ENGINE VERSION
vz79cf4danb0njb3mg9p8vwpv *  node1     Ready     Active
                            Leader   19.03.0-beta2
trmx4medo7ffaxy0e3z5xkhvt  node2     Ready     Active
                            19.03.0-beta2
[node1] (local) root@192.168.0.37 ~
$ [node1] (local) root@192.168.0.37 ~
# completely the user's responsibilities.
#
# The PWD team.
#####
[node2] (local) root@192.168.0.38 ~
$ docker swarm join --token SWMTKN-1-1u4oi5o8db4831ipn37cyof8attas3h0xt2bf2y0fdf2my0id5-6xje7khf
2.168.0.37:2377
This node joined a swarm as a worker.
```

Figure 2.8: Adding worker node in Docker swarm

Once we have the cluster ready, we can start running services on it. To register a new service, we go to the console of a manager and execute the following:

```
$ docker service create --replicas 1 --name helloworld alpine ping www.google.com
```

Where --name is the name of the service and --replicas is the number of tasks of this service that we want to create. Now we can see a list of all the services in the cluster:

```
$ docker service create --replicas 1 --name helloworld alpine ping www.google.com
4b5umtqhuuhjxj88xv3yk91520
overall progress: 1 out of 1 tasks
1/1: running
verify: Service converged
[node1] (local) root@192.168.0.23 ~
$ docker service ls
ID          NAME      MODE      REPLICAS      IMAGE
GE          PORTS
4b5umtqhuuhjx  helloworld replicated  1/1          alpine:latest
```

Figure 2.9: Creating replica service in Docker swarm

With the following commands, we can see all the tasks that are running for this service, scale the number of replicas, and check which nodes are running the new replicas:

```
$ docker service ps helloworld
```

```
$ docker service scale helloworld=4
```

```
$ docker service ps helloworld
```

In the following screenshot, we can see how scaling a service with 4 replicas in 2 nodes.

```

$ docker service ps helloworld
ID          NAME      IMAGE      NODE      DESIRED STATE
CURRENT STATE      ERROR      PORTS
mf33ajplktg3    helloworld.1  alpine:latest  node2      Running
    Running about a minute ago
[node1] (local) root@192.168.0.23 ~
$ docker service scale helloworld=4
helloworld scaled to 4
[node1] (local) root@192.168.0.23 ~
$ docker service ps helloworld
ID          NAME      IMAGE      NODE      DESIRED STATE
CURRENT STATE      ERROR      PORTS
mf33ajplktg3    helloworld.1  alpine:latest  node2      Running
    Running 2 minutes ago
tidzecyit0z8     helloworld.2  alpine:latest  node2      Running
    Running 7 seconds ago
fqmov6k5j2wa     helloworld.3  alpine:latest  node1      Running
    Running 7 seconds ago
3qx814c75j1v     helloworld.4  alpine:latest  node1      Running
    Running 7 seconds ago

```

Figure 2.10: Scaling replicas number in Docker swarm

In this lab, you will play around with the container orchestration features of Docker. You will deploy a simple application to a single host and learn how that works. Then, you will configure a Docker swarm mode and learn to deploy the same simple application across multiple hosts. You will then see how to scale the application and move the workload across different hosts easily.

<https://training.play-with-docker.com/orchestration-hol>

The screenshot shows a web-based Docker playground. At the top, there's a navigation bar with 'Play with Docker classroom' and 'About'. Below it, the title 'Section 2: Configure Swarm Mode' is displayed. To the left, there's a terminal window showing the command `docker run -dt ubuntu sleep infinity` being run. The output of this command is shown below the terminal. To the right, there's a large text area containing instructions and examples related to Docker swarm mode, including commands like `docker run` and `docker service`.

Figure 2.11: Configuring swarm mode

In this lab, you will begin to explore running multiple services as a single stack with Docker swarm <https://training.play-with-docker.com/ops-s1-swarm-intro>

Play with Docker classroom

Swarm Mode Introduction for IT Pros

Sep 12, 2017 • @jimmodified

So far we have explored using single instances of containers running on a single host, much like a developer might do when working on a single service application or like an IT administrator might do on a test rig. Production applications are usually much more complex and this single server model will not work to coordinate 10s or 100s of containers and the network connections amongst them, not to mention the need to ensure availability and the ability to scale.

For real applications IT users and app teams need more sophisticated tools. Docker supplies two such tools: **Docker Compose** and **Docker Swarm Mode**. The two tools have some similarities but some important differences:

- **Compose** is used to control multiple containers on a single system. Much like the *Dockerfile* we looked at to build an image, there is a text file that describes the application: which images to use, how many instances, the network connections, etc. But **Compose** only runs on a single system so while it is useful, we are going to skip **Compose**¹ and go straight to **Docker Swarm Mode**.
- **Swarm Mode** tells Docker that you will be running many Docker engines and you want to coordinate operations across all of them. Swarm mode combines the

```

If the commandline doesn't appear in the terminal, make sure popups are enabled or try resizing the browser window.

tive Leader 19.03.0-beta2
[node1] (local) root@192.168.0.13 ~
$ docker node ls
ID HOSTNAME STATUS AV
AIBILITY MANAGER STATUS ENGINE VERSION AC
ce2zbe5272nk58ibdatf103zz * node1 Ready
tive Leader 19.03.0-beta2
[node1] (local) root@192.168.0.13 ~
$ [node2] (local) root@192.168.0.12 ~

WARNING!!!!
# This is a sandbox environment. Using personal credentials
# is HIGHLY! discouraged. Any consequences of doing so are
# completely the user's responsibilites.
#
# The PWD team.
#####
[node2] (local) root@192.168.0.12 ~
$ [node1] (local) root@192.168.0.13 ~

```

Figure 2.12: Swarm mode introduction for it pros

In this section, we have analyzed Docker swarm as a Docker native clusters management tool. For its original design, it is more about a scheduler than a tool that manages the life cycle of our applications. From the point of view of filtering, tags, and the scheduler, we have seen that it offers many options and very flexible, so it is a problem that solves quite well without losing its compatibility with the remote API of the Docker engine.

OpenShift container platform

Red Hat OpenShift container platform helps organizations easily develop, deploy, and manage existing and new applications in physical, virtual, and public cloud infrastructures.

The last version is available at <https://www.redhat.com/en/openshift-4/details>. In addition to using a newer version of Kubernetes, CRI-O, the new version will be released as the default runtime container. CRI-O <https://cri-o.io/> is the new container runtime designed for Kubernetes, which allows executing any container image that follows the Open Container Initiative (OCI) standard and is compatible with Docker images.

OpenShift as Platform as a Service

A Platform as a Service (PaaS) is a type of cloud service in which a platform is delivered where services and functions of an application are executed with a minimum of operational load for the developer. In a PaaS model, the developer focuses mainly on its code, leaving the hard platform work to define where the application is executed, high availability, or maintenance of the operating system, among others.

RedHat OpenShift proposes a complete platform of containers integrating Docker, Kubernetes as native technologies of execution and container orchestration with a series of special functions to manage permissions, storage, application life cycle, and other functions of the enterprise base in Red Hat Enterprise Linux.

DevOps with OpenShift

Another important advantage is that OpenShift offers a common platform and a group of tools for the development teams and operations of your company. Fostering a common and continuous work culture and dynamics for both teams in the development and maintenance of applications. This allows us to eliminate processes and slow or manual routes, increasing the pace of work according to the needs of your company.

OKD <https://www.okd.io> is a distribution of Kubernetes optimized for continuous application development and multi-tenant deployment. OKD also serves as the upstream codebase upon which Red Hat OpenShift Online and Red Hat OpenShift container platforms are built. For more information check documentation <https://docs.okd.io/index.html>

Using the container orchestration system of the Kubernetes project, OpenShift has a set of additional functionalities that make it the ideal platform for the integration of DevOps environments:

It allows the construction of traditional applications, as well as oriented to the cloud. A set of integrated middleware platforms for the development and deployment of applications.

It allows managing the life cycle of applications based on containers.

It includes tools for converting source code into running application thanks to the source-to-image process.

Being a DevOps tool offers organizations mechanisms to improve communication between development and operations, as well as eliminate integration barriers between both departments, thanks to the following features:

Self-provisioning: The main problem that development finds is the waiting time since the application architect has developed the diagram until the developer can start writing code. OpenShift will allow you to reduce this process to just a few minutes since with a simple command from the developer, and you can provide the hardware, the software, as well as the network.

Multi-language: Allows the use of different languages, platforms, databases ... allowing developers to use all the possibilities of Docker. Therefore, OpenShift will not limit your users to develop in a single platform, but it gives you the power to choose the language you prefer.

Automation: OpenShift offers you automated systems to manage the life cycle of applications in the most effective way.

Collaboration: OpenShift allows the management of roles that will enable a set of operations or others, to different users, within the same project. As a simple example, you can allow a user of the QA/Testing team to monitor the status of a development project, and when the application is running in that environment, promote it to the QA environment or even to Production.

Application portability: Being built on Docker containers, this allows our application to be migrated to any system that uses Docker.

Open source: with all the possibilities and advantages that free software provides us.

Scalable: Allows applications to scale easily and automatically.

From the operations point of view, it has the next features:

Promotion between environments: Thanks to the automation processes of OpenShift, the promotion between environments can be delegated to those responsible for the DEV and QA teams.

Scalability and planning: When operations or development teams request the platform to scale an application horizontally, the platform is responsible for deploying in the nodes that are available the number of replicas that have been indicated. Once the number of replicas has been modified, OpenShift updates the HA-Proxy records, thus enabling load balancing. On the other hand, OpenShift has a Scheduler that plans the deployment of the applications between the available nodes, differentiating zones and regions, for adequate deployment of environments.

Teams and roles: OpenShift allows the management of roles that will enable a set of operations or others, to different users, within the same project. For example, we can allow a user of the QA/Testing team to monitor the status of a development project, and when the application is running in that environment, promote it to the QA or production environment.

As we have seen, the additional features offered by the Red Hat OpenShift Container platform, beyond the Docker containerization engine and the Kubernetes orchestration, make this technology ideal for the integration and promotion of DevOps environments. Integral middleware platforms facilitate deployment tasks, facilitating life cycles, construction, and delivery agile and secure. Processes such as Source-to-Image allow continuous development and delivery (CI/CD), converting the source code into functional applications in execution.

There are 4 versions of OpenShift:

OpenShift Origin (<https://github.com/openshift>). This version allows you to have an OpenShift cluster managed by Red Hat to deploy your applications.

OpenShift online <https://www.openshift.com/products/online>. It allows you to create and execute applications in the public cloud offered by Red Hat. You can test OpenShift online if you login with RedHat account credentials: <https://manage.openshift.com/>

OpenShift dedicated <https://www.openshift.com/products/dedicated> : allows you to have an OpenShift cluster managed by Red Hat to deploy your applications.

OpenShift container platform <https://www.openshift.com/products/container-platform> : allows you to have an OpenShift cluster in your own infrastructure managed by Red Hat.

For deploying an OpenShift cluster instance in a local environment, we have 2 options:

Run OKD in a Container following documentation from docs.okd.io site https://docs.okd.io/latest/getting_started/administrators.html#running-in-a-docker-container

Try out a fully functioning OKD instance with an integrated container registry, running locally on your machine with minishift. This tool allows you to build a cluster of a single node on a virtual machine. Here <https://www.okd.io/minishift>. You can find all the necessary documentation about it and the instructions to start it. It is inspired by minikube, the solution to run Kubernetes locally.

OpenShift core items

In this section, we will see an introduction to the existing basic resources to understand how PaaS works. We will use the following image corresponding to the console to describe these resources in addition to other information present in it:

The screenshot shows the OpenShift Overview project for a 'demo' application. The left sidebar lists 'Overview', 'Applications', 'Builds', 'Resources', 'Storage', 'Monitoring', and 'Catalog'. The main content area is titled 'demo' and contains sections for 'DEPLOYMENT CONFIG', 'CONTAINERS', 'NETWORKING', and 'BUILDS'. In the 'DEPLOYMENT CONFIG' section, a blue box highlights 'DEPLOYMENT CONFIG demo, #1'. In the 'CONTAINERS' section, a red box highlights 'Image: myproject/demo d397407 229.5 MB' and 'Build: demo, #1'. An orange box highlights 'Ports: 8080/TCP'. A blue circle icon indicates '2 pods'. In the 'NETWORKING' section, a red box highlights 'Service - Internal Traffic demo' and '8080/TCP (8080-tcp) → 8080'. A green box highlights 'Routes - External Traffic http://demo-myproject.2886795287-80-rhsummit1.environments.katacoda.com'. Below the networking section, a green box highlights 'Route demo, target port 8080-tcp'. The 'BUILDS' section shows a build named 'demo' created 3 minutes ago.

Figure 2.13: OpenShift Overview project

In the previous image, we can distinguish the following elements:

Route (green): the route is the URL associated with a service so that we can invoke it, a route can be open to the Internet or only for internal use. In this case, the path is welcome-PHP demo.cloudapps.example.com. Typically the default routes are generated as [name of service] - [name of project].[Domain of PaaS]

Exposed ports (orange): these are the ports that the service exposes and with which it is mapped in our container. In this case, port 8080 is mapped to port 8080 of the service.

Service (red): each service represents each of the applications that we have in our project, the service will be the entry point to the application and will expose some ports for its communication.

Deployment (blue): represents the deployment configuration for that application, it indicates the number of instances, their configuration, which container image to use, scaling parameters. The displayed number represents the number of times we perform a new deployment.

Pod (blue circle): each instance that we want to execute from our application will be executed in a different pod; thus, if, for example, we have two instances, we will have two instances of the same Docker image running each one in its own pod.

In the build section, we can see information related to the content displayed in the pod, such as the Docker image that is being used, the git repository from which it was generated.

The screenshot shows the 'Builds' section for the 'demo' application. It displays a single build named 'demo' created 3 minutes ago. The build status is 'app' (highlighted in blue). There are tabs for 'History', 'Configuration' (underlined), 'Environment', and 'Events'. The 'Details' section shows build configuration details: 'Build Strategy: Source', 'Source Repo: https://github.com/openshift/django-ex.git', 'Source Ref: master', 'Builder Image: openshift/python:3.6', 'Output To: myproject/demo:latest', and 'Run Policy: Serial'. The 'Triggers' section shows triggers for the build: 'Config Change For: Build config demo openshift/python:3.6', 'New Image For: Generic Webhook URL: https://2886795287-8443-frugo02.environments.katacoda.com', 'GitHub Webhook URL: https://2886795287-8443-frugo02.environments.katacoda.com', and 'Manual (CLI): oc start-build demo -n myproject'.

Figure 2.14: OpenShift deployment configuration

In the deployment section, we can see deployment configuration; in it, we can find information corresponding to:

Configuration of creation and destruction of instances during deployment.

Configuration of the number of replicas.

The Docker image that is being used and its version.

Environment variables are internal to the container.

Status of the current deployment.

Result of the last deployments.

OpenShift's deployment configs are constantly monitoring to see the number of pods running.

Deployments > demo > #1

The screenshot shows the deployment configuration details for 'demo-1'. The status is 'Active' with 'demo' as the deployment config. The reason for the status is 'config change'. The 'Selectors' field includes 'deployment=demo-1' and 'deploymentconfig=demo'. The 'Replicas' field shows '2 current / 2 desired' with a pencil icon. A large blue circle contains the number '2' and the word 'pods', with up and down arrows to its right for scaling.

Figure 2.15: OpenShift details deployment configuration

To verify a number of replicas, click the pod's number in the circle next to the arrows. You should see a list with your pods by scrolling to the bottom of the web console:

Containers

demo

- Image: myproject/demo e1d501d 229.5 MiB
- Build: demo, #1
- Source: Merge pull request #137 from danmcp/patch-1 0905223 authored by Ben Parees
- Ports: 8080/TCP

Volumes

Add Storage | Add Config Files

Pods

Name	Status	Containers Ready	Container Restarts	Age
demo-1-qsqx6	Running	1/1	0	6 minutes
demo-1-7l45h	Running	1/1	0	8 minutes

Figure 2.16: OpenShift pods running

You can see that we have 2 replicas. Application scaling can happen extremely quickly because OpenShift is just launching new instances of an existing image, especially if that image is already cached on the node.

Next, we will discuss the different alternatives when building and deploying applications in OpenShift. In the documentation, we can see that highlights the following options to make a build:

Source-to-Image (S2I): This is a framework that allows, taking the source code of your application as input, generating an image that executes the source code. This will create a Docker image for each version of your source code that you want to run.

https://docs.openshift.com/enterprise/3.1/creating_images/s2i.html

Docker builds: This strategy will execute a Docker build and wait for the Docker image to be generated in the registry to use it.

<https://docs.openshift.com/enterprise/3.1/architecture/coreconcepts/buildsandimagestreams.html#docker-build>

Custom strategy: this strategy is that you create a Docker image yourself, what you do is precisely to execute the process of building your application in another image, which will be the one that is deployed.

<https://docs.openshift.com/enterprise/3.1/architecture/coreconcepts/buildsandimagestreams.html#custom-build>

These different construction strategies allow us to generate Docker images to leave in the registry to execute them later. What we must highlight from these three alternatives is that every time we want to deploy a new version of our code it will be necessary to create a new Docker image in the registry, so if we make many deployments it can be an overload for our system from the point of view. view of processing capacity and storage.

OpenShift also is pre-loaded with Source-to-Image (S2I) builders for Java, JavaScript (Node.js), Perl, PHP, Python, and Ruby.

The screenshot shows the OpenShift web interface for a project named 'myproject'. On the left, there's a sidebar with options like Overview, Applications, Builds, Resources, Storage, Monitoring, and Catalog. The main area is titled 'Select an item to add to the current project' and has tabs for All, Languages, Databases, Middleware, CI/CD, and Other. Under Languages, there are categories for All, Java, JavaScript, .NET, Perl, Ruby, PHP, and Python. Below these, there are sections for .NET (with .NET Core, .NET Core + PostgreSQL (Persistent), .NET Core Example, .NET Core Runtime Example), PHP (with CakePHP + MySQL), and Python (with Python). Each section includes a small icon and a link to a template.

Figure 2.17: OpenShift pre-loaded templates

Learning scenarios

In the URL <https://learn.openshift.com>, you can find interactive learning scenarios that provide you with a pre-configured OpenShift instance accessible from your browser without any downloads or configuration. Use it to experiment, learn OpenShift, and see how we can help solve real-world problems.

The image displays a grid of six learning scenario cards, each with a red 'START COURSE' button at the bottom. The cards are arranged in two rows of three. The top row contains: 'Foundations of OpenShift', 'Building Applications On OpenShift', and 'Subsystems, Components, and Internals'. The bottom row contains: 'OpenShift Playgrounds', 'Service Mesh Workshop with Istio', and 'Building Operators on OpenShift'.

Figure 2.18: OpenShift learning scenarios

Conclusion

In this URL <https://container.training>, we can find other resources for learning related to presentations and videos made in different conferences.

Containers allow you to deploy applications faster, start and stop them faster and take better advantage of hardware resources. Virtual machines allow us to create completely isolated complete systems, with greater control over the environment and mixing host and guest operating systems.

Each technology has its applications and its advantages according to the needs and circumstances of each development. Currently, containers in general and Docker in particular, are becoming an indispensable technology and are increasingly used for more, not only to deploy applications in production but also to create replicable development environments among all members of a team, ensure that the applications are going to execute the same in all environments (development, testing, and production). Some people say that, in the medium term, most developers will use Docker to develop and deploy applications. Let's see what happens, but Docker undoubtedly offers very important advantages in all phases of software development.

CHAPTER 3

Managing Containers and Docker Images

This chapter covers topics related to how Docker manages images and containers, the main commands used for generating our images from Dockerfile, and how we can optimize our docker images, minimizing the size of images and, in consequence reducing the attack surface.

The images are read-only templates that we can use as a base to launch containers. This means that what we do in the container only persists in that container; these modifications are not done in the image. If we want to have a personalized image, we must create it for our future containers. In this chapter, we will see how we can create an image from a container that we have already customized.

Structure

Managing Docker images

Dockerfile commands

Managing Docker containers

Inspecting Docker containers

Optimizing Docker images

Objectives

Understanding the concept of managing Docker container and images

Understanding Dockerfile commands and best practices for optimizing

Knowing about Inspecting Docker containers

Knowing about optimizing Docker images

Managing Docker images

A Docker image is a frozen or hibernating system, which is in a read-only mode. The images are formed by layers that are mounted on top of each other. All layers are mounted in read-only mode, except the last layer that is mounted as read/write and gives rise to the container.

A container is nothing more than a running image, so when you run the image, Docker adds a layer on the image in a read/write mode. When the container stops or is deleted, Docker automatically removes the read/write layer leaving the image in its original state. This allows you to reuse the same image in several environments.

Introducing Docker images

If we download an image using the pull command and then save it, we can see that an image is a set of directories and files with a specific structure, where each folder refers to one of the layers of which it is formed the picture. Within each layer, there are some files to reference the said layer and a compressed file with the file system that will form the image.

```
$ docker pull alpine
```

```
$ docker save alpine -o alpine.tar
```

Figure 3.1: Executing pull and save commands

When an image is extracted and constructed so that it can be usable, what is done is to decompress the content of each layer in order from the last one, which corresponds to the base image, thus generating a file system whose content is built or modified incrementally with each layer.

The last layer that is mounted in read/write mode is the one that differentiates one container from another or any container from its base image. All the structures that are made on a container that they do are add new data or modify the existing data in the last layer. When a container is removed, the writing layer is also deleted, but the base image remains unchanged.

An image is a permanently stored instance of a container. The Docker images command shows you the images on your system. You can assign multiple aliases (including names and tags) to the same image whenever it is useful.

If a container runs in the background, it can continue to run after the docker run command ends. You can see containers running with the docker ps command.

You can see saved containers that are no longer running with the docker ps -a command. You can restart a container; you can use the docker start command. You can stop a running container with the docker stop command.

Docker layers

Layers are like Git confirmations and store the difference between the previous and current version of the image. And like Git commits, they are useful if you share them with other repositories or images. In fact, when you request an image of a record, download only the layers that you don't have downloaded to your machine locally. This way, it is much more efficient to share images.

The layers use space, and the more layers you have, the thicker the final image will be. Git repositories are similar in this regard. Because Git stores all changes between commits, the size of your repository increases with the number of layers.

We can see the layers of an image with the command \$docker image history : . In this example, we are obtaining layers from Python image.

IMAGE	CREATED	CREATED BY	SIZE
COMMENT			
14a2caeca327	17 hours ago	/bin/sh -c #(nop) CMD ["python3"]	0B
<missing>	17 hours ago	/bin/sh -c set -ex; wget -O get-pip.py "\$P..."	6.24MB
<missing>	17 hours ago	/bin/sh -c #(nop) ENV PYTHON_GET_PIP_SHA256...	0B
<missing>	17 hours ago	/bin/sh -c #(nop) ENV PYTHON_GET_PIP_URL=ht...	0B
<missing>	17 hours ago	/bin/sh -c #(nop) ENV PYTHON_PIP_VERSION=19...	0B
<missing>	2 weeks ago	/bin/sh -c cd /usr/local/bin && ln -s idle3...	32B
<missing>	2 weeks ago	/bin/sh -c set -ex && wget -O python.tar.x...	91.2MB
<missing>	2 weeks ago	/bin/sh -c #(nop) ENV PYTHON_VERSION=3.7.4	0B
<missing>	2 weeks ago	/bin/sh -c #(nop) ENV GPG_KEY=0D96DF4D4110E...	0B
<missing>	2 weeks ago	/bin/sh -c apt-get update && apt-get install...	17.1MB
<missing>	2 weeks ago	/bin/sh -c #(nop) ENV LANG=C.UTF-8	0B

Figure 3.2: Docker layers in Python image

Another way to obtain the layers of an image is through the MicroBadger online service that shows the contents of Docker's public images, including metadata and information about the layers that make up the images. <https://github.com/microscaling/microbadger>.

The following image shows the information of a Python-based image: <https://microbadger.com/images/python>:

Metadata from image python

Last inspected 28 minutes ago.

Versions ▾

Tags	latest	buster	3	3.7	3.7.4	3.7.4-buster	3.7-buster	3-buster																																																
Created	July 30, 2019 at 02:33 AM																																																							
ID	008b021b6899																																																							
Download Size	330.4 MB																																																							
Labels	No labels																																																							
Layers	18																																																							
<table border="1"> <tr> <td>297.7 MB</td> <td>buildpack-deps</td> <td>testing</td> <td>buster</td> <td>What's this?</td> <td>–</td> <td></td> <td></td> <td></td> </tr> <tr> <td>48.0 MB</td> <td>ADD file:2cddee716e84c40540a69c48051bd2dcf6cd3bd02a3...</td> <td colspan="6">⊕</td> </tr> <tr> <td>7.4 MB</td> <td>CMD ["bash"]</td> <td colspan="6"></td> </tr> <tr> <td>9.5 MB</td> <td>RUN apt-get update && apt-get install -y --no-instal...</td> <td colspan="6">⊕</td> </tr> <tr> <td>49.4 MB</td> <td>RUN set -ex; if ! command -v gpg > /dev/null; then ...</td> <td colspan="6">⊕</td> </tr> <tr> <td></td> <td>RUN apt-get update && apt-get install -y --no-instal...</td> <td colspan="6">⊕</td> </tr> </table>								297.7 MB	buildpack-deps	testing	buster	What's this?	–				48.0 MB	ADD file:2cddee716e84c40540a69c48051bd2dcf6cd3bd02a3...	⊕						7.4 MB	CMD ["bash"]							9.5 MB	RUN apt-get update && apt-get install -y --no-instal...	⊕						49.4 MB	RUN set -ex; if ! command -v gpg > /dev/null; then ...	⊕							RUN apt-get update && apt-get install -y --no-instal...	⊕					
297.7 MB	buildpack-deps	testing	buster	What's this?	–																																																			
48.0 MB	ADD file:2cddee716e84c40540a69c48051bd2dcf6cd3bd02a3...	⊕																																																						
7.4 MB	CMD ["bash"]																																																							
9.5 MB	RUN apt-get update && apt-get install -y --no-instal...	⊕																																																						
49.4 MB	RUN set -ex; if ! command -v gpg > /dev/null; then ...	⊕																																																						
	RUN apt-get update && apt-get install -y --no-instal...	⊕																																																						

Figure 3.3: Metadata from Python image in MicroBadger service

Image tags

Image tags allow identifying the versions of the images, when listing images, they are listed with their associated tag. In the previous Python example, the tag I had downloaded is the latest.

A good example that serves to understand tags and see how layers work is the following. If we go to the Python Docker hub page https://hub.docker.com/_/python we can see the tags in the description:

Supported tags and respective Dockerfile links

Simple Tags

- 3.8.0b2-buster, 3.8-rc-buster, rc-buster
- 3.8.0b2-slim-buster, 3.8-rc-slim-buster, rc-slim-buster, 3.8.0b2-slim, 3.8-rc-slim, rc-slim
- 3.8.0b2-alpine3.10, 3.8-rc-alpine3.10, rc-alpine3.10, 3.8.0b2-alpine, 3.8-rc-alpine, rc-alpine
- 3.8.0b2-windowsservercore-ltsc2016, 3.8-rc-windowsservercore-ltsc2016, rc-windowsservercore-ltsc2016
- 3.8.0b2-windowsservercore-1803, 3.8-rc-windowsservercore-1803, rc-windowsservercore-1803
- 3.8.0b2-windowsservercore-1809, 3.8-rc-windowsservercore-1809, rc-windowsservercore-1809
- 3.7.4-buster, 3.7-buster, 3-buster, buster
- 3.7.4-slim-buster, 3.7-slim-buster, 3-slim-buster, slim-buster, 3.7.4-slim, 3.7-slim, 3-slim, slim

Figure 3.4: Tags from the official Python image in Docker hub

With the docker pull command, we download a specific image, and previously we have downloaded them when creating a container. In this example, we are downloading a specific version for Python with the command \$ docker image pull python:3.8-rc-alpine3.10 :

```
$ docker image pull python:3.8-rc-alpine3.10
3.8-rc-alpine3.10: Pulling from library/python
050382585609: Pull complete
dac2222ca532: Pull complete
a5a8a13f5210: Pull complete
48ed6fe4c480: Pull complete
f5c21fef32f5: Pull complete
Digest: sha256:e686f6b5cf95f23bbb19d4f38a9f541abfdca0f7f6be6b74fbf862db068793be
Status: Downloaded newer image for python:3.8-rc-alpine3.10
docker.io/library/python:3.8-rc-alpine3.10
[node1] (local) root@192.168.0.43 ~
$ docker images
REPOSITORY          TAG           IMAGE ID      CREATED        SIZE
python              latest        14a2caeca327    17 hours ago   918MB
python              3.8-rc-alpine3.10  70dal2d86711    17 hours ago   109MB
```

Figure 3.5: Tags when pulling Python image

Design considerations for Docker images

An image is made up of layers that are mounted one on top of the other. The last layer is mounted as read/write mode and gives a place to the container. Layers use the copy on the write pattern.

When a new container is created from an image, all layers in the image are read-only, and a thin read-write layer is added above. All changes made to the specific container are stored in that layer.

The original layered organization and copy-on-write strategy promote some of the best practices for creating and shaping Docker images.

Minimalist images: Docker images get huge benefits from the point of view of stability, security, and loading time while smaller. If you need to solve problems related to development, you can always install tools in a container.

Choosing a base image: The choice of the base image is an important decision. It can contain many layers and add many capacities, but also a lot of weight. The quality of the image and the author are also critical. There are official images for many distributions, programming languages, databases, and runtime environments available in the Docker hub.

Dockerfile commands

One of the nice things about containers built using the automated build approach is that Docker hub will show you the Dockerfile used to build the container, which provides some level of transparency over what you're downloading (and the syntax is relatively basic, so easy to see what's going on). You still need trusting the Docker hub and the sources of data used to build the container.

Images are created using a series of commands, called instructions. The instructions are placed in the Dockerfile file, which is basically a text file that contains a collection of changes in the root file system and the corresponding execution parameters for use within a container later. Docker will read this file when the image creation process begins and executes the instructions one by one.

The result will be the final image. Each instruction creates a new layer in the image. That image layer then becomes the parent of the layer created by the next instruction.

What is a Dockerfile?

Dockerfiles are scripts containing declared commands that will be executed successively, in the order given, by Docker to automatically create a new Docker image. These help greatly during deployments.

The Dockerfile allows you to build an image, and this image can be uploaded to a registry so that it can be downloaded to the servers you use to deploy your application. The recommended way is not to make changes to the container and then to commit them, but to write a Dockerfile.

Building images from Dockerfile

The docker build command instructs the daemon to create an image, for which the corresponding Dockerfile must be available. If the user has not created the image but takes it from a repository in Docker hub, then the docker pull command is executed.

When the daemon is instructed to start a container with the docker run command, the program first checks if the required image is stored locally. If yes, the container starts (solid line). It can also happen that the daemon does not find the image, from which it extracts one directly from the repository.

A Docker image corresponds to the information needed to start a container, and basically, it consists of a file system and other metadata such as the commands to be executed, the environment variables, the container volumes, and the ports used by our container. The build of an image ends once the Docker image is uploaded to a Docker registry, at which point the application deployment period begins.

The main steps to create an image from a Dockerfile file are:

Create a new directory that contains the file, with the script and other files that were necessary to create the image.

Create the content.

Build the image using the docker build command.

The syntax for the command is:

```
$ docker build [options] [Dockerfile_path]
```

The most common options are:

-t, name [: tag]: Create an image with the name and label specified from the instructions indicated in the file. It is a highly recommended option.

-no-cache: By default, Docker caches recently performed actions. If it is the case that we execute a docker build several times, Docker will check if the file contains the same instructions and, if so, will not generate a new image. To generate a new image by omitting the cache, we will always use this option.

-pull: Docker will only download the image specified in the FROM expression if it does not exist. To force you to download the new version of the

image, we will use this option.

-quiet: By default, the entire creation process, the executed commands, and their output are displayed. Using this option will only show the identifier of the created image.

Dockerfiles always start with the definition of a base image using the FROM instruction. The recommended way to build an image is to use a Dockerfile, a set of instructions that indicate how to build a Docker image. The main instructions that can be used in a Dockerfile are:

FROM : Establishes the base image of our container.

RUN : Allows you to execute a command in the context of the image.

CMD : Establishes the command that the container executes on startup.

EXPOSE: You can define ports where the container accepts connections.

ENV var = value: To define environment variables.

COPY

: Allows you to copy files and directories to the file system of the container

VOLUME : To define volumes in the container.

For a complete list of available instructions, you can check the official documentation <https://docs.docker.com/engine/reference/builder/>.

The COPY, ADD and RUN instructions add a new layer to your image. The following Dockerfile example creates two layers, one for each RUN command.

FROM ubuntu

RUN apt-get update

RUN apt-get install vim

A good practice is to combine several RUN instructions in a single line, so we would only have one layer.

FROM ubuntu

RUN apt-get update && apt-get install vim

Among the instructions found in this file, we can highlight:

FROM instruction: The FROM instruction sets the base image for the following instructions. The image can be any local or public image. If the image is not found locally, the Docker compilation command will attempt to download the image from the public record. The tag or tag command is optional, so if it is not specified, the latest tag is assumed by default.

FROM :

RUN instruction: The RUN instruction will execute any command in a new layer at the top of the current image and confirm this image. The generated image will be used for the next instruction in the Docker file. The RUN instruction has two forms:

RUN

RUN ["executable", "arg1", "arg2" ...]

The RUN instruction is only interpreted and used at the time the docker build command is used to create an image. The purpose of the RUN instructions is to execute commands that modify the image in some way. For example, you can install software packages or create a configuration file that becomes part of the image. In this example, a file is created at compile time and then viewed with Docker RUN :

FROM fedora:latest

MAINTAINER maintainer

RUN echo "This container was built on \$(date)." > /tmp/built.txt

ENTRYPOINT ["cat", "/tmp/built.txt"]

When the Docker compilation is executed, the command reads the current date and time and sends it to the file /tmp/build.txt . Because the command was executed at compile-time, the exact same date is displayed each time you perform a docker run command.

```
$ docker build -t fedora_image.
```

```
$ docker run fedora_image
```

```
$ docker build -t fedora_image .
Sending build context to Docker daemon 1.747MB
Step 1/4 : FROM fedora:latest
--> 2b74bf3d2430
Step 2/4 : MAINTAINER maintainer
--> Using cache
--> 471a4d43a7c5
Step 3/4 : RUN echo "This container was built on $(date)." > /tmp/built.txt
--> Using cache
--> 2c90489b36b6
Step 4/4 : ENTRYPOINT ["cat","/tmp/built.txt"]
--> Using cache
--> 8a83454955f9
Successfully built 8a83454955f9
Successfully tagged fedora_image:latest
[node1] (local) root@192.168.0.43 ~
$ docker run fedora_image
This container was built on Tue Jul 30 18:19:06 UTC 2019.
```

Figure 3.6: Executing docker build and docker run with fedora image

In this Dockerfile, we indicate in the script which base image we are going to use through FROM , then with RUN , we point the commands to be executed, and with CMD , we say the default command.

```
FROM ubuntu:latest
```

```
RUN apt-get -y update; \
```

```
apt-get -y upgrade; \
```

```
apt-get -y install apt-utils \
```

```
vim \
```

```
htop;
```

```
RUN apt-get -y install dstat
```

```
CMD ["bash"]
```

Once the file is created, we save it. Now we make the construction of the image. The creation of the image is executed by the docker engine, which receives all the information from the environment; therefore, it is advisable to save the Dockerfile in an empty directory and add the necessary files for the creation of the image. The docker build command executes the instructions of a Dockerfile line by line and displays the results on the screen.

```
$ docker build -t "test_dockerfile".
```

Figure 3.7: Executing docker build with the ubuntu image

The previous image is only the beginning part of the execution of the docker build. Now we can see in the list of available images and with the run command, we create a container from the image:

```
$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
test_dockerfile  latest   e22b016f25f7  3 seconds ago  176MB
ubuntu          latest   3556258649b2  3 days ago   64.2MB
[node1] (local) root@192.168.0.33 ~/ubuntutest
$ docker run -dti --name mycontainer e22b016f25f7
aca0408912343d042b3352146dd87e891179941498237e220a5d254175288625
[node1] (local) root@192.168.0.33 ~/ubuntutest
$ docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS
PORTS      NAMES
aca040891234      e22b016f25f7      "bash"      12 seconds ago      Up 10 sec
onds      mycontainer
```

Figure 3.8: Executing docker image and docker ps commands

And we access it to verify that, effectively, the installed programs are available; in this case, we are executing dstat command inside the container to get statistics about processes.

```
$ docker exec -i -t mycontainer /bin/bash
```

```

$ docker exec -i -t mycontainer /bin/bash
root@aca040891234:/# dstat
You did not select any stats, using -cdngy by default.
--total-cpu-usage-- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai stl| read writ recv send| in out| int csw
 31 17 52 0 0| 174k 2591k| 0 0 | 0 0| 12k 47k
 28 27 45 0 0| 0 0 | 0 0 | 0 0| 27k 87k
 31 28 41 0 0| 0 32k | 0 0 | 0 0| 28k 100k
 28 25 47 0 0| 0 4096B| 0 0 | 0 0| 28k 89k
 30 30 39 0 0| 0 396k | 0 0 | 0 0| 28k 100k
 29 26 45 0 0| 0 296k | 0 0 | 0 0| 28k 91k
 31 29 41 0 0| 0 4096B| 0 0 | 0 0| 27k 99k
 28 26 45 0 0| 0 48k | 0 0 | 0 0| 27k 89k

```

Figure 3.9: Executing dstat command inside the container

Best practices writing Dockerfiles

Docker exposes in its documentation a section of good practices for writing Dockerfiles. These practices will help us create images more efficiently, modularly, and with less effort.

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

<https://docs.docker.com/engine/reference/builder>

Among the best practices to create optimized Docker images, we can highlight:

Run only one process per container: Following the practice of a single process per container allows us to make decoupled applications and reuse containers more easily, that are easier to scale, and results in more decoupled systems. This also allows us to use container links or other container networking techniques.

Reduce the size of your images: A Docker image should only contain what is strictly necessary to run your application. In order to reduce complexity, dependencies, image size, build times of an image, you should avoid installing packages just because they can be useful for debugging a container. As an example, do not include text editors in your images. Another very practical option is the use of small base images, for example, using alpine.

Do not assume that our containers are always running: Containers should be treated as immutable entities, which means we should not modify them while they are running. The recommendation at this point is to modify the Dockerfile, reconstruct the image, and lift a container with that updated image. Therefore, it is recommended to handle data and execution configurations outside the container and, therefore, its image. To handle data isolated from the execution of the container, we can use Docker volumes.

Use official images of the Docker Hub, instead of writing ours from scratch: These images are maintained by who are the companies that provide this software. We can also use ONBUILD images to simplify the process of creating our images.

Minimize the number of layers of our images using the image cache: Docker uses Union Filesystems to store images. This means that each image is made from a base image plus a collection of differences that add the required changes. Each difference represents an additional layer in an image. This has a direct impact on how we write our Dockerfile and the directives we use.

Reduce the size of images to a minimum: A Docker image should only contain what is strictly necessary to run the application. In order to reduce complexity, dependencies, image size, and image construction times, you should avoid installing packages just because they can be useful for debugging a container. A good option is the use of smaller base images, for example, making use of the alpine distribution as a base image.

Use reduced base images: Dockerfile files start from a base through the FROM statement, which is at the beginning of the file. The rest of the instructions will depend on this instruction. For example, if we start from a CentOS image, the package manager to be used in the RUN instruction will be yum instead of apt-get. This is because, in that base image from which we started, we don't have apt-get available as a package manager. For example, an image based on alpine Linux <https://alpinelinux.org> is lighter than other official ones.

Managing Docker containers

The Docker Hub <https://hub.docker.com> provides you and your organization with a place to host and deliver images. You can configure the Docker hub repositories in two ways: Repositories, which allow us to upload and update the images whenever we want from the docker daemon and automatic images that allow us to configure a GitHub or BitBucket account that triggers the reconstruction of an image when any changes are made to the repository.

Search and execute a Docker image

If you are interested in trying out a new software application or looking for a new one that serves a particular purpose, Docker images can be an easy way to experiment without installing and configuring anything on your machine. Suppose you are interested in trying Node.js. We could perform a search with the search command.

```
$ docker search node
```

Figure 3.10: Executing docker search node command

Another way we can use to search for an image is through the DockerHub interface.

The screenshot shows the DockerHub search interface. At the top, there's a search bar with 'node' typed in. Below it, the main content area displays search results for 'node'. The first result is 'node' by 'node', which is described as 'Updated a few seconds ago'. It's labeled as an 'OFFICIAL IMAGE' with '10M+' downloads and '7.7K' stars. A brief description states: 'Node.js is a JavaScript-based platform for server-side and networking applications.' Below this, there are filters for 'Docker Certified' (unchecked), 'Verified Publisher' (unchecked), and 'Official Images' (unchecked). There are also sections for 'Images' (with 'Verified Publisher' and 'Official Images' filters) and 'Categories' (with 'Analytics' filter). The second result shown is 'mongo-express'.

Figure 3.11: Searching node application in DockerHub

Once we have downloaded the node image, we will launch a container based on that image and interact with the command line of that container with the docker run command.

```
$ docker run [options] [image] [commands] [arguments]
```

When executing the docker run command, we must specify an image that we will use as a base when launching the container, and another point is

that the options can replace almost all the default values configured in the execution. When starting a container, we have several configuration parameters:

- i allows establishing a connection with the standard input
- t manage a pseudo TTY
- d run the container in background mode.
- a associate standard input or output to the open session
- cpus number of CPUs assigned
- ip assigns an IP address
- mac-address assigns a special MAC address to the container.
- m set a memory limit for that container (usually a few megabytes)
- name assigns a name to the container
- p publish container ports in the assigned network
- rm stopping the container will be automatically deleted.
- tmpfs mount a directory in tmpfs mode (temporary to be deleted, no persistence)
- v mount a directory in the container with persistence, it can be a real computer folder or a Docker volume.

To execute this image, we execute the run command and can be told to do it interactively using the -t and -i flags. The -t flag creates a tty device (a terminal) and the -i flag specifies that this session is interactive:

```
$ docker run -t -i node /bin/bash
```

```
root@0f3ebbafe91c:/# node
```

```
Welcome to Node.js v12.7.0.
```

```
Type ".help" for more information.
```

```
process.version
```

```
'v12.7.0'
```

Here we are already interacting with the container, that numbering that you see after the root is the id of the container, we can update the system, install packages, etc. This container also has an IP with which we can interact with the container. We can visualize the containers that we have in execution with the docker ps command.

Executing a container in background mode

To execute a container in background mode, use the --detach or -d option . The -d option allows you to indicate that it runs in the background (usually as a service daemon process).

```
$ docker container run --detach -ti --name mypython python:latest /bin/bash
```



Figure 3.12: Executing Python container in detach mode

Inspecting Docker containers

Although Docker commands give you access to information about images and containers, sometimes you want to get more information about the metadata of these objects. The docker inspect command gives access to the metadata of a Docker image in JSON format. The syntax of the command is as follows:

```
$ docker inspect [OPTIONS] CONTAINER|IMAGE|TASK [CONTAINER |IMAGE|TASK...]
```

<https://docs.docker.com/engine/reference/commandline/inspect>

You can run docker inspect command in a running container or in one that is no longer running but has not been deleted. In other words, any container that can you see with the commands docker ps or docker ps -a .

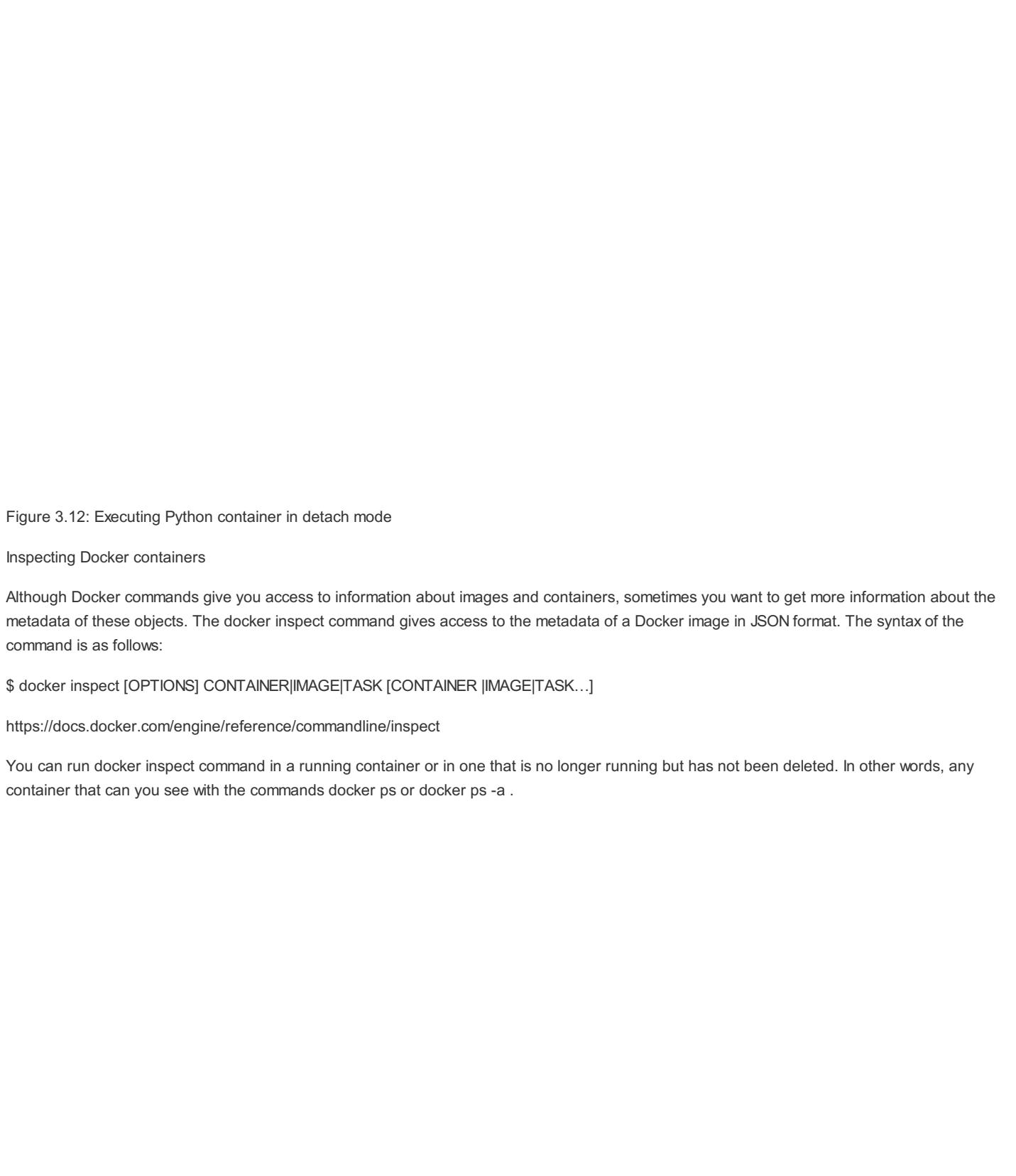


Figure 3.13: Inspecting Python image

docker inspect provides a series of options that allow you to identify specific attributes with the --format option. For example, you can verify the IP address configured for your container.

You can inspect images and containers by name or ID. In this command, we use the docker inspect command to obtain the IP address of the container.

```
$ docker inspect --format '{{.NetworkSettings.IPAddress}}'
```

Figure 3.14: Inspecting Docker container for searching IP address

To verify that the IP address is active, you can channel that output to the ping command as follows:

```
$ ping -c 1 $(docker inspect --format='{{.NetworkSettings.IPAddress}}' )
```

Figure 3.15: Inspecting Docker container for checking ping command

The following command obtains the IP addresses of all running containers on the Docker host. Keep in mind that the ping command only accepts one IP address, and it is necessary to pass an additional argument to xargs -l1 to indicate that you execute the command for each line individually.

```
$ docker ps -q | xargs docker inspect --format = '{{.NetworkSettings.IPAddress}}' | xargs -l1 ping -c1
```

Figure 3.16: Inspecting Docker container for checking IP addresses

The previous command can be summarized in the following steps:

Gets the identifiers of the running containers.

We execute the docker inspect command from the identifiers obtained in the previous step.

Get the IP address of each container and run ping for each.

We can check packages installed in a Docker container. For example, if we are using an image-based in Ubuntu, we can use the command dpkg -l for checking packages installed. First, we need to find the id of the container that is running.

```
$ docker exec dpkg -l
```

\$ docker exec 5f12dcc38753 dpkg -l				
Desired=Unknown/Install/Remove/Purge/Hold				
Status=Not/Inst/Conf-files/Unpacked/half-inst/trig-aWait/Trig-pend				
Err?=(none)/Reinst-required (Status,Err: uppercase=bad)				
Name	Version	Architecture	Description	
ii adduser	3.115	all	add and remove users and groups	
ii apt	1.4.8	amd64	commandline package manager	
ii base-files	9.9+deb9u2	amd64	Debian base system miscellaneous f	iles
ii base-passwd	3.5.43	amd64	Debian base system master password	and group files
ii bash	4.4-5	amd64	GNU Bourne Again SHell	
ii bsdutils	1:2.29.2-1	amd64	basic utilities from 4.4BSD-Lite	
ii coreutils	8.26-3	amd64	GNU core utilities	
ii dash	0.5.8-2.4	amd64	POSIX-compliant shell	
ii debconf	1.5.61	all	Debian configuration management sy	stem
ii debian-archive-keyring	2017.5	all	GnuPG archive keys of the Debian a	rchive

Figure 3.17: Inspecting packages in Docker container-based in Ubuntu image

Optimizing Docker images

Optimizing space and reducing container size is essential to create efficient container environments. Docker images are generated from a series of layers. Each layer represents an instruction in the Dockerfile file. All layers, except the last one, are read-only.

If we think that Docker is designed to be able to mount a big number of containers, both space and speed are key factors in a development environment and much more productive. One way to optimize images is to use as few layers as possible. The following set of instructions occupies 4 layers.

RUN apt-get update -y

RUN apt-get install -y curl

RUN apt-get install -y postgresql

RUN apt-get install -y postgresql-client

During construction, whenever possible, Docker tends to reuse the layers of an image of previous construction, ignoring a step that could be costly. We need to consider these particular use cases:

Place the Dockerfile instructions that tend to change in the final part of the file. This way, Docker can reuse the previous layers.

We could group instructions that are repeated in the same layer. This is due to many reasons, from reusing the layer, to maintaining the same context, since it may be necessary to maintain a context between two layers. To avoid this, a clear example of this is the apt command, which usually requires an update of repositories and previous packages.

In this case, the same command with fewer lines only occupies one layer:

```
RUN apt-get update -y && \
apt-get install -y curl postgresql postgresql-client
```

Docker's cache

The construction of a Docker image from a Dockerfile can be an expensive process since it can involve the installation of a big number of libraries, and at the same time, it is a repetitive process because successive builds of the same Dockerfile are very similar to each other. That is why Docker introduces the concept of cache to optimize the image building process.

Each time an image is reconstructed from a Dockerfile, Docker checks if the current instruction has been executed correctly and, therefore, has the results of the instruction available in the cache. If the results are correct and are cached, Docker, by default, uses the instruction's cached data and reuses it with the new compilation.

In the first way, the command is executed in a shell, specifically the shell `/bin/sh -c`. The second way is useful in cases where the base image does not have a shell. Docker uses a cache for the construction of the images, in this way, in case the process fails somewhere in between, the next execution is able to reuse the partial compilations and continue from the point where it failed.

The first optimization that the Docker cache does is the download of the base image of our Dockerfile. Docker downloads the base image if it is not already downloaded to the machine that builds. This optimization seems obvious since these images can be hundreds of MB in size, but you have to be careful, because if the remote version of the image changes, Docker will continue to use the local version. Therefore, if we want to run our Dockerfile with the new version of the base image, we must do a manual Docker pull of the base image.

As we mentioned earlier, a Docker image has an internal structure, quite similar to a git repository. What we know as commits in git we call layers of an image in Docker. Therefore, an image is a succession of layers in a Docker registry, where each layer stores the differences from the previous layer. This concept is important in order to optimize our Dockerfiles.

For now, it will be enough to know that each instruction of our Dockerfile will create one and only one layer of our image. Therefore, the Docker cache works at the instruction level. In other words, if a Dockerfile line does not change, instead of recomputing it, Docker assumes that the layer that generates that instruction is the same as the previous Dockerfile execution. Therefore, if we have an instruction such as:

```
RUN apt-get update && apt-get install -y git
```

When executing 2 successive builds, the `apt-get` commands will not be executed, but the layer that generated the first build will be reused.

Therefore, even before executing the second build, there is a new version of the Git package, the image built from this Dockerfile will have the previous version of Git, which was installed in the first build of this Dockerfile. We can deactivate the use of the cache running `docker build --no-cache`.

It is important to highlight the following aspects about the Docker cache:

The Docker cache is local, that is, if it is the first time you build a Dockerfile on a given machine, all Dockerfile instructions will be executed, even if the image has already been built in a Docker registry.

If an instruction has changed and you cannot use the cache, the cache is invalidated, and the following Dockerfile instructions will be executed without using the cache.

The behavior of the ADD and COPY instructions are different in terms of the behavior of the cache. Although these instructions do not change, they invalidate the cache if the content of the files being copied has been modified.

Finally, if for some reason you want to build without using the cache, you can use the `--no-cache = true` flag for that purpose.

When creating our image from the Dockerfile file, there is an interesting feature that we can use to reconstruct the image using the Docker cache, so that a certain layer associated with a command is only rebuilt if the command has changed. The cache will be invalidated in these situations:

When the `docker build` command is executed with the `--no-cache` flag.

When a command that can be cached is provided, such as executing `apt-get update` command.

If the context content has changed, the first ADD instruction invalidates the cache for all the following instructions in the Docker file.

For example, to force a complete reconstruction of the image without using the cache, we can use the `--no-cache` flag.

Docker caches each layer to accelerate the creation of images. The cache is used for instruction if:

The previous instruction was found in the cache.

There is a layer in the cache that has the same instruction.

In addition, in the case of the COPY and ADD instructions, the cache will be invalidated if the checksum or metadata of any of the files in that layer has changed.

Docker build optimization

When we are working on our local machine, Docker has the layers of all previous docker build executions that we have made. In other words, when we work on our local machine, the Docker cache is initialized, remember the execution of previous docker build instructions, and as we have seen in previous lessons, it greatly optimizes the generation of images. However, this is not usually the case in continuous integration environments, where normally each integration task (either the build of an image or the execution of tests) is executed on a new independent machine created for this purpose and destroyed when the homework ends.

This is because reusing the same machines between continuous integration tasks can lead to the appearance of unwanted states in the machine or the accumulation of garbage that will end up saturating the resources of said machine.

Consequently, when a task of continuous integration is executed, it is quite frequent that the Docker cache is not initialized, which implies that the times to build our images can grow enormously. This is a major problem because continuous integration tasks should execute the faster, the better.

Luckily, Docker has a solution to this problem. When we execute a docker build, we can indicate that it uses the cache of an already created image. In other words, if I am building the python:latest image, I can reuse the cache of the latest version of python:latest by running these commands:

```
$ docker pull python:latest
```

```
$ docker build -t python:latest --cache-from=python:latest
```

Building an application with Node.js

In this example, we will develop a web page with Node.js that will be served by a web server that will run in a Docker container. In Docker, we also have the option of joining multiple layers in a structure called multi-stage. In this example, we build a Node.js container with an Express-based application.

index.js

```
const express = require('express')

const app = express()

app.get('/', (req, res) => res.send('Hello World!'))

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

package.json

{

"name": "hello-world",

"version": "1.0.0",

"main": "index.js",

"dependencies": {

"express": "^4.16.2"

},

"scripts": {

```
"start": "node index.js"
```

```
}
```

```
}
```

In this example, we use a node-based image, and we package this application with the following Dockerfile, where we execute the npm install command from the package.json and index.js files:

```
FROM node:8

EXPOSE 3000

WORKDIR /app

COPY package.json index.js./

RUN npm install

CMD ["npm", "start"]
```

Next, we create our image, from the directory where we have saved the Dockerfile. We can build the image with the following docker build command:

```
$ docker build -t node-app
```

We can verify that in our local environment we have the image we just created with the command: docker image ls

We execute the image for creating the container with the application running on port 3000:

```
$ docker run -d -p 3000:3000 -ti --rm --init node-app
```

```
hello-world@1.0.0 start /app
```

```
node index.js
```

Example app listening on port 3000!

In the Dockerfile file, there are 2 COPY and RUN commands that generate two additional layers to the base image. The resulting image has five new layers, one for each statement in its Dockerfile file. We can see the different layers with the docker history command.

We can verify that in our local environment, we have the container we just created with commands docker container ls and docker history node-app .

Reducing image size with multistage

Now let's test the construction of the Dockerfile through multiple stages. We will use the same Dockerfile, but now we rewrite it with multi-stage mode. The main difference from the previous one is that in this case, we are using the FROM node:8 statement twice.

```
FROM node:8 as build
```

```
WORKDIR /app
```

```
COPY package.json index.js./
```

```
RUN npm install
```

```
FROM node:8
```

```
COPY --from=build /app /
```

```
EXPOSE 3000
```

```
CMD ["index.js"]
```

The first part of the Dockerfile creates three layers. The layers are fused and copied in the second and last stage. Two more layers are added above the image for a total of 3 layers. When executing the build and history commands, we can see how the image generated with multi-stage is

smaller with the following commands.

In the construction of the second image (the one that aims to run in production) takes the executable of the application of the previous image (from = build), named as build.

```
$ docker build -t node-multi-stage.
```

Figure 3.18: Building a Docker image with multistage mode

With the following command, we can verify the size of the image using node-multi-stage.

```
$ docker images | grep node-
```

```
node-multi-stage latest 70d53ac4b571 3 minutes ago 897MB
```

For more information, you can see the docker documentation about multistage-build: <https://docs.docker.com/develop/develop-images/multistage-build/> .

Reducing image size with alpine Linux

Alpine Linux-based images <https://docs.docker.com/samples/library/alpine> are also quite popular, as they produce the smallest images to run applications with few resources at the memory and disk space level. In this sense, images based on this distribution are much faster to download and configure.

For example, we can use Python 3.6 in Alpine, creating a Dockerfile like this:

```
FROM python:3.6-alpine
```

```
CMD ["python3.6"]
```

Build and run this Dockerfile offers you in a Python 3.6 Shell. This image is excellent if you run a Python application that does not require many system-level dependencies.

In our Node application, the distribution of alpine-Linux offers us the possibility to reduce the size of the image if we use it as a base image.

```
FROM node:8 as build
```

```
WORKDIR /app
```

```
COPY package.json index.js./
```

```
RUN npm install
```

```
FROM node:8-alpine
```

```
COPY --from=build /app /
```

```
EXPOSE 3000
```

```
CMD ["npm", "start"]
```

```
$ docker build -t node-alpine.
```

```
$ docker images | grep node-alpine
```

\$ docker images grep node				
node-alpine	latest	448a795c1bbf	20 seconds ago	68.4MB
node-multi-stage	latest	70d53ac4b571	8 minutes ago	897MB
node	8	8e45c884a32e	3 weeks ago	895MB
node	8-alpine	e08ba08cf75a	8 weeks ago	66.7MB

Figure 3.19: Checking the image size of Docker container-based in node image

We can verify that the image size has been reduced from 897MB to 68.4MB, and we can see that the application can be executed in the same way as before.

```
$ docker run -p 3000:3000 -ti --rm --init node-alpine
```

Example app listening on port 3000!

We can even access the shell of the image; here, we can use the sh command instead of bash to be based on the Alpine Linux distribution. We will have to replace the container identifier with the one we have obtained.

```
$ docker exec -tish
```

Distroless images

From the point of view of security, if we get smaller and specialized images, focusing on only one function or application, attack vectors are reduced, as well as network traffic and, therefore, the risk. An attacker in our system, the first thing he will try to achieve is a shell, from here to execute what interests him, pivot or execute lateral movements (to obtain the maximum possible information of the system), data exfiltration, search for persistence and so on.

This also drastically reduces system updates and, therefore, complete maintenance of all mounted architecture. And this is where images without a system or distroless images play an important factor from the point of view of security.

Distroless images contain only the application and its dependencies at runtime. They do not contain package management applications or programs that we normally find in a standard Linux distribution.

In the following GitHub repository, we can find the source code of the project: <https://github.com/GoogleCloudPlatform/distroless>.

The Google Container Tools project hosts a series of Docker images oriented to certain programming languages without an operating system; that is, they do not contain any distribution, and all the images contain are the files needed to run the application.

By executing the following commands, we can see how the difference in sizes between an official image (98.6MB) and another that uses a reduced version (50.9MB), is almost 50 MB.

```
$ docker pull gcr.io/distroless/python3
```

```
$ docker pull python:alpine
```

```
$ docker pull gcr.io/distroless/python3
Using default tag: latest
latest: Pulling from distroless/python3
e8d8785a314f: Pull complete
e005d777a298: Pull complete
3e010093287c: Pull complete
609f69c3154c: Pull complete
Digest: sha256:b83bd4dc7c34d1c3alb8400474163fccfe5b9110d0d1cb12b48e1786473d5ba2
Status: Downloaded newer image for gcr.io/distroless/python3:latest
gcr.io/distroless/python3:latest
[node1] (local) root@192.168.0.43 ~
$ docker pull python:alpine
alpine: Pulling from library/python
050382585609: Pull complete
dac2222ca532: Pull complete
29a7fe408caa: Pull complete
6ad337b9b53f: Pull complete
31d663a76478: Pull complete
Digest: sha256:d22196e0ced4a0fd44916e3ff4aea00565260f66a3d0d26f5551b8fdbd833423
Status: Downloaded newer image for python:alpine
docker.io/library/python:alpine
```

Figure 3.20: Downloading Python Docker image-based in distroless and alpine

```
$ docker image ls
```

Figure 3.21: Comparing Docker image size Python distroless vs.Python alpine

This not only saves us disk space and network traffic but also improves security. Not having libraries that we are not going to need, we reduce security risks and vulnerability scanner alerts due to obsolete or vulnerable versions.

Among the images currently available, we can highlight:

```
gcr.io/distroless/python2.7
gcr.io/distroless/python3
gcr.io/distroless/nodejs
gcr.io/distroless/java
```

In this URL <https://github.com/GoogleContainerTools/distroless/tree/master/examples/python3> , you can see an example of the construction of our Dockerfile for an application based on Python 3 using distroless approach.

```

1 FROM python:3-slim AS build-env
2 ADD . /app
3 WORKDIR /app
4
5 FROM gcr.io/distroless/python3
6 COPY --from=build-env /app /app
7 WORKDIR /app

```



Figure 3.22: Dockerfile example using Python3 distroless image

But since Distroless is a simplified version of the original operating system, there are no additional binaries, and it is not possible to run a bash or sh to get a shell.

The only binary that I could execute for this image is the Python interpreter. In this way, we have improved both the size of the 900 MB image that occupies the official image to 50 MB that occupies the distroless version and improved its security, because an attacker if you manage to exploit the application and access the container, you cannot access a shell to execute commands, you will only have access to the binaries that have the image installed.

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python3	latest	f6d85470e9f6	21 minutes ago	52.5MB
<none>	<none>	801bdc56f2ed	23 minutes ago	52.5MB
python	3-slim	ca7f9e245002	2 weeks ago	143MB
python	latest	a4cc999cf2aa	2 weeks ago	929MB
gcr.io/distroless/python3	latest	b31fdb42763	49 years ago	50.9MB

```

$ docker run -it python bash
root@a4c6043a9fef:/# exit
exit
[node1] (local) root@192.168.0.8 ~
$ docker run -it python3 bash
/usr/bin/python3.5: can't open file 'bash': [Errno 2] No such file or directory
[node1] (local) root@192.168.0.8 ~

```

Figure 3.23: Executing container-based in Python3 distrolessDocker image

We can conclude that fewer binaries mean smaller image sizes and greater security.

In this example, we are going to use the image of Nodejs gcr.io/distroless/nodejs . The construction of our Dockerfile for the application of Node.js using this library would be as follows:

```
FROM node:8 as build
```

```
WORKDIR /app
```

```
COPY package.json index.js./
```

```
RUN npm install  
FROM gcr.io/distroless/nodejs  
COPY --from=build /app /  
EXPOSE 3000  
CMD ["index.js"]
```

With the previous Dockerfile file, we go on to build the image, and we can run the application and see that everything works normally:

```
$ docker build -t node-distroless.  
$ docker images | grep node
```

We can verify that with a distroless image, the image size has been reduced from 897MB to 74.4MB, and we can see that the application can be executed in the same way as before.

Figure 3.24: Comparing Docker image size node distroless vs. node alpine

In this way, we have improved both the size of the image and its security. If an attacker manages to exploit the application and gains access to the container, he will not be able to access a shell to execute commands, and he will only have access to the binaries that have the image installed. We can conclude that fewer binaries mean smaller image sizes and greater security.

Conclusion

Docker images are based on a layered file system that offers many advantages and benefits for use cases for which containers are designed, such as being lightweight and sharing common parts that many containers can deploy and run on the same machine economically.

You only need to understand the principles and mechanisms to use Docker images effectively. Docker provides several commands to get an idea of what images are available now and how they are structured.

CHAPTER 4

Getting Started with Docker Security

This chapter covers topics like security best practices and other aspects like Docker capabilities, which containers leverage in order to provide more features, such as the privileged container. Also, we will review the Docker content trust and Docker registry that provide a secure way to upload our images in the Docker hub platform and private registry. While Docker provides a central registry to store public images, you may not want your images to be accessible to the world. In this case, you must use a private registry.

Docker gives an approach to run applications safely segregated in a holder, bundled with every one of its conditions and libraries. Since your application can generally be kept running with the environment, the build image, testing, and organization is less complex, as your build will be completely compact and prepared to keep running in any environment.

New IT professionals with knowledge about containers need to have a good understanding of container security. As a result, those who manage and secure container applications need to learn those skills quickly.

Structure

Docker security principles

Security best practices

Docker capabilities

Docker content trust

Docker registry

Objectives

Understanding Docker security principles and security best practices

Understanding Docker capabilities and Docker content trust

Knowing about Docker registry

Knowing about creating a Docker registry in localhost

Docker security principles

From the security point of view, Docker Containers use the resources of the host machine, but they have their own runtime environment. They have a reduced version of the user space of the operating system. This means that you can secure a container in a very similar way to what you would use in a real machine since, in the background, it is almost as if it were a real machine.

A container cannot access other containers or the underlying operating system (except for the storage volumes to which you give permission) and will communicate with other networks and containers, with the specific network configuration that you want to grant.

Keep in mind that a container isolates you as much as a virtual machine, but it is much less heavy, requires less configuration, and can also open wormholes to specific folders if we need it, with total security.

Even when processes isolated in containers run in the same kernel, Docker uses a series of isolation techniques to protect them from each other. The most important is the central functions of the Linux kernel, such as Cgroups and namespaces. The distribution of system resources (memory, CPU, bandwidth) takes place by means of a Cgroup mechanism that guarantees that each container can only consume the quota reserved for it.

With this approach, the containers do not offer the same degree of isolation that can be achieved with virtual machines. In the case that an attacker was made with a virtual machine, it would hardly have any chance to attack the core of the underlying host system. Containers, on the other hand, as encapsulated instances of a common host kernel, provide a greater margin of freedom for any type of attack.

Despite the isolation techniques that have been described, from the containers it is possible to reach important secondary kernel systems such as Cgroups or the kernel interfaces in the /sys and /proc directories, and these offer attackers the ability to dodge the functions Host security And since all containers run on the same host system in the same user namespace, a container that has been assigned admin (root) rights also keeps them in interaction with the host kernel.

The Docker daemon, responsible for the management of the container in the host system, also enjoys root rights, so a user with access to the daemon automatically gains access to all the directories to which access has been authorized as well as the possibility to communicate with a REST API using HTTP. This is why, in its documentation, Docker recommends providing access to the daemon only to trusted users.

The development team behind Docker is also aware of these security problems, considering them an obstacle to the consolidation of this technology in production systems. Along with the fundamental isolation techniques of the Linux kernel, the latest versions of the engine Docker, therefore, support the AppArmor, SELinux and Seccomp frameworks, which would act as a kind of firewall for the kernel resources:

AppArmor allows regulating the access rights of the containers to the file system.

SELinux presents a complex system of rules with which you can implement access controls to the kernel resources.

Seccomp (Secure Computing Mode) monitors system calls.

Adding to these frameworks, Docker also uses the so-called Linux capabilities with which you can limit the root rights with which the Docker Engine starts the containers.

Some software weaknesses contained in application components that are expanded through Docker's registry are also a source of objections. In principle, there are no restrictions regarding the creation and publication of images in the Docker hub, and this is what carries the risk of introducing malicious code in a system by downloading an image. Therefore, before deploying an application, Docker users should always make sure that the complete code supplied by an image to run containers comes from a reliable source.

The use of Docker brings benefits to developers, testers, and system administrators. In the case of developers, the use of Docker makes it possible to focus on code generation and not worry about the different characteristics that the development and production environment can have. On the other hand, since it is very easy to manage containers and one of their main characteristics is that they are very light, they are very suitable to deploy a testing environment where you can do the testing. Finally, it also brings advantages to system administrators, since the deployment of applications can be done more easily, without the need to use virtual machines.

The great advantage of container-based virtualization is that applications with different requirements can run isolated from each other without having to assume the overhead of a separate guest system. For this, container technology takes advantage of two basic functions of the Linux kernel: the control groups (Cgroups) and the kernel namespaces.

Namespaces also provide isolation for processes and mount points. In this way, processes that run in a container cannot interact or see processes that run in another container. The isolation of the mounting points implies that they cannot interact with the mounting points in another container.

Control groups (Cgroups) are a feature of the Linux kernel that facilitates the limitation of the use of resources at the level of CPU and memory that a container can use. This ensures that each container gets only the resources it really needs.

Docker daemon attack surface

While Docker facilitates virtualization work, it is sometimes possible to forget the security implications of the execution of Docker containers. From a security point of view, it is important to keep in mind that Docker requires root privileges to function normally.

Docker Daemon is the main process that manages the life cycle of containers and needs root privileges to run. Unfortunately, since the Docker daemon runs with root privileges, it also presents an attack vector. For more details, we can get more information in the official documentation <https://docs.docker.com/engine/security>.

The Docker daemon is responsible for creating and managing containers, which includes creating file systems, assigning IP addresses, routing packets, process management, and many more tasks that require administrator privileges. Therefore, it is essential to start the daemon as a user administrator.

Among the main actions we can perform on the containers, we can highlight operations such as starting new containers, stopping running containers as well as reconfiguring the running containers with new commands. It is also possible to extract confidential information, such as passwords and certificates.

One of Docker's ultimate goals is to be able to run even the daemon as a non-root user, without affecting its functionality, and delegate operations that require root (such as file and network system operations) to a dedicated thread with elevated privileges.

If you want to expose the Docker port to the outside (to make use of the remote API), it is recommended to ensure that only trusted clients have access allowed. A simple way is to secure Docker with SSL and certificates using HTTPS. You can find ways to configure this at <https://docs.docker.com/articles/https>.

Security best practices

In the following list, we will summarize the best security practices when running Docker:

It is advisable to run the daemon Docker process on a dedicated server and isolated from other machines.

The best option to run the Docker daemon is to use a machine with a Unix operating system.

Special care must be taken to link certain Docker host directories as volumes since it is possible for a container to gain full read and write access and perform irreversible operations on these directories.

From the point of view of communications, the best option is to use SSL-based authentication.

Avoid running processes with root privileges inside the containers.

We could study the option of enabling specific security profiles such as AppArmor and SELinux on the host Docker.

Unlike virtual machines, all containers share the host Docker kernel. Therefore, it is important to have the kernel updated with the latest security patches.

From the point of view of system administrators, they could follow best practice recommendations for Linux system administration. The following best practices can help create services with more security and improve container security:

Do not run the software as root.

Disable SETUID permissions.

Use the -cap-drop and -cap-add flags to remove and add capabilities in the container.

Use the --cap-add flag, only for the capabilities you really need.

One application per container, microservice oriented approach.

If you are going to share secrets, it is advisable not to use environment variables or run containers in privileged mode.

It is also advisable to check the users who have access to the Docker host.

It is important to have Docker updated to the latest version in order to ensure that all holes have been solved and that it also has the latest features that Docker is incorporating.

Update the Linux kernel in the Docker host. One of the most vulnerable components in container management is the kernel, as it is shared among all containers. Therefore, special care should be taken to keep the kernel in its latest update when available.

In the following sections, we are going to analyze some best practices with more details. First, we are going to check the default user within a container.

Execution with a non-root user

By default, containers run with root privileges. If we execute the following commands, we see that the default user is root.

In this command we are executing alpine container for checking root user:

```
$ docker run -v /bin:/host/bin --it --rm alpine sh
```

This is the execution of the previous command:

```
$ docker run -v /bin:/host/bin -it --rm alpine sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
e7c96db7181b: Pull complete
Digest: sha256:769fddc7cc2f0a1c35abb2f91432e8beecf83916c421420e6a6da9f8975464b6
Status: Downloaded newer image for alpine:latest
/ # whoami
root
/ # id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
```

Figure 4.1: Executing a Docker container with default root user

At this point, from a security point of view, it is important to configure the namespaces to limit access to the container. While the container engine must be run with the root user, it is not a good practice for the containers to do so, and it is necessary to create a user for each running container.

The containers are executed by default with the root user, so root privileges are available within the container. The security solution is to indicate in the creation of the image in the Dockerfile, the user who wants to be able to execute.

You can add the user inside the Dockerfile with the following commands:

```
RUN useradd
```

```
USER
```

We could include in the Dockerfile the information about the user with the previous commands:

```
FROM python:latest
```

```
RUN useradd -s /bin/bash unix_user
```

```
USER unix_user
```

```
ENTRYPOINT ["bin/bash"]
```

We build the image with the command:

```
$ docker image build -t python_image
```

When executing the container with the interactive option (-i), we see how the user corresponds to the one we have declared in the Dockerfile.

```
$ docker run -t python_image
```

In this screenshot, we can see the content from /etc/password file after executing the previous command:

```
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534:::nonexistent:/usr/sbin/nologin
unix_user:x:1000:1000::/home/unix_user:/bin/bash
```

Figure 4.2: Inspecting the file /etc/passwd

In this way, when inspecting the file etc/passwd , we see how the user is added inside the container.

Start containers in read-only mode

Best practice recommendations for the administration of Linux systems include the application of the principle of minimum privilege. For this, flags such as read-only can be applied when executing a container.

Limiting the use of the file system against writing helps prevent a potential attacker from writing to the container.

To do this, use the docker run command with the read-only flag:

```
$ docker run -d --read-only python sh
```

In the following screenshot, we can see the result of executing the previous command:

```
$ docker run -it --read-only python sh
Unable to find image 'python:latest' locally
latest: Pulling from library/python
85b1f47fba49: Pull complete
ba6bd283713a: Pull complete
817c8cd48a09: Pull complete
47cc0ed96dc3: Pull complete
4a36819a59dc: Pull complete
db9a0221399f: Pull complete
7a511a7689b6: Pull complete
1223757f6914: Pull complete
Digest: sha256:59d8481f4b2d21f2ac6623e986b4e91fa704112df3e7d9ddbe7315d4
a153ef5
Status: Downloaded newer image for python:latest
# touch file
touch: cannot touch 'file': Read-only file system
```

Figure 4.3: Executing container with read-only mode

When executing the container with this flag, if we try to create a file, it returns the message cannot touch the file: read-only filesystem.

The main disadvantage with the read-only parameter is that most applications need to write files in directories such as / tmp and will not work in a read-only environment. In these cases, we could use folders and files in which the application needs to write access and use volumes to mount only those files.

If the container needs to write to the filesystem, a volume can be provided to avoid errors and, in addition, make changes persistent once the container dies. In the case of temporary files, it is recommended to use Docker volumes.

A volume is a directory that is separate from the root filesystem of the container and is managed directly by the daemon Docker process and can be shared between containers.

In this example, we are running a MySQL container and configure it as read-only, with the exception of /var/lib/mysql and /tmp directories. This means that the only location where data can be written into the container is in these directories. Any other location inside the container will not allow you to write anything on it.

To do this, we can run the MySQL container in combination with other parameters like MYSQLROOTPASSWORD and define a volume with -v flag:

```
$ docker run --name mysql --read-only -v /var/lib/mysql -v /tmp -d -e MYSQLROOTPASSWORD=password mysql
```

In this screenshot we can see the output when trying to create a file inside a container with read-only mode:

```
$ docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED      STATUS
PORTS             NAMES
79b33d74c408      mysql      "docker-entrypoint.s..."   20 seconds ago   Up 19 seconds
3306/tcp, 33060/tcp  mysql
[node1] (local) root@192.168.0.38 ~
$ docker exec mysql touch /opt/filename
touch: cannot touch '/opt/filename': Read-only file system
```

Figure 4.4: Executing a MySQL container with a volume

When executing the container and trying to write outside the /tmp directory, we get the error message read-only filesystem.

When working with volumes, we could use the flag: ro when indicating the route where we declare the volume:

```
$ docker run -v $(pwd):/pwd:ro debian touch /pwd/x
```

In the following screenshot, we can see the result of executing the previous command:

Figure 4.5: Executing container with a volume in read-only mode

At this point, we have reviewed how to start a container and mount a volume in read-only mode.

Disable setuid and setgid permissions

The set user ID (setuid) and set group ID (setgid) bits are special permissions that are used to access directories and files in the operating system by users that do not have root permissions.

The main problem with these bits is that they can be exploited by attackers. At this point, the best practice is to disable the setuid rights by adding these lines in the Dockerfile. With the following command, the setuid and setgid permissions are deleted during the image construction phase using the Dockerfile.

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

This command performs a search for executables and removes any setuid and setgid permission from any user. It is important to apply this command carefully since it is surely necessary that legitimate programs require such permissions and therefore do not remain unusable within the container.

With the following command you can disable setuid and setgid bits when you start a Docker container:

```
$ docker run -d --cap-drop SETGID --cap-drop SETUID
```

With the previous command, you have disabled setuid and setgid capabilities when running a specific Docker container.

Verifying images with content trust

DOCKERCONTENTTRUST environment variable allows verifying that images you download from the Docker registry or Docker hub are trusted and signed. In this way, you can defend against poisoned or injected images. For enabling this feature, you need to export this variable with export DOCKERCONTENTTRUST=1 .

In the following command, we are downloading an image from the Docker hub, additionally verifying the image hash:

```
$ docker pull someimage@sha256:a25306f3850e1bd44541976aa7b5fd0a29be
```

In the previous command, we are checking the SHA256 hash of the image file system manifest, where a manifest is a metadata file that describes the constituent parts of a Docker image. The manifest file contains a list of all the image layers identified by the hash, so if you can verify that the manifest has not been modified, you can download and trust all layers in a secure way, even over untrust channels like HTTP.

Resource limitation

The Docker service, by default, all containers share host machine resources equitably. This means that there is no preference between containers when it comes to consuming resources. One of the problems that may arise, and that is why there are applications or software aimed at monitoring the infrastructure or cluster of containers to be able to see, in a granular way, which containers may be affecting the stability of the entire infrastructure and following this, cause a denial of service to the host machine and therefore affect the entire container ecosystem, preventing its normal operation.

A possible solution to resource problems that may arise is to limit each of the containers using the command:

```
$ docker run [OPTIONS] [IMAGE] [COMMAND] [ARG]
```

The previous command has different configuration parameters that allow both to limit the use of resources and to prioritize in case it is needed at a specific time in order to meet the needs of different users.

The following command shows information about options available related to CPU, devices, and memory:

```
$ docker run --help | grep 'cpu|device|memory'
```

In the following screenshot, we can see the result of executing the previous command:

Figure 4.6: Command options for limiting resources in containers

At this point, we have reviewed the different options available to the command to meet performance needs, such as CPU, memory, and read/write speed.

Docker capabilities

The containers, by default, boot with a series of limited Linux privileges. One of the strengths is the granularity that these systems have in order to provide permissions, in case it requires the situation, and to carry out the functionality that needed to be executed, without providing root permissions.

In the same way that different privileges can be added, by default, the ideal in terms of security is always to apply as restricted or as minimalist as possible. In other words, do not provide permissions until it is shown that they are necessary to be able to execute the different functionalities required. Therefore, by applying this concept, the exposure of the container is minimized.

Docker capabilities allow us to manage what permissions a process has to access the kernel and allow segregating root user privileges to limit actions that can be accessed with privileges. The Docker container engine uses part of Linux's capabilities and is essential for managing security contexts.

The capabilities provide a tool with which to design a more advanced security strategy with different privilege levels.

In this URL <http://man7.org/linux/man-pages/man7/capabilities.7.html>, you can check the man pages for Linux capabilities.

Capabilities allow us to manage what permissions a process has access to the parts of the Kernel, regardless of the user who launches it. They allow segregating root user privileges to limit actions that can be accessed with privileges. Different container engines use part of the capabilities of Linux and are essential for managing security contexts.

Linux performs a granular division of capabilities so that the privileges associated with the superuser can be enabled and disabled independently. Some example capabilities are:

CAP_SYSLOG: For modifying the behavior of the kernel log.

CAP_NETADMIN: For modifying the network configuration.

CAPSYSMODULE: For managing kernel modules.

CAPSYSRAWIO: For modifying the kernel memory.

CAPSYSNICE: For modifying the priority of the processes.

CAPSYSTIME: For modifying the system clock.

CAPSYSTT_CONFIG: For configuring TTY devices.

CAPAUDITCONTROL: For configuring the audit subsystem.

Thanks to this granularity that the capabilities provide and the fact of not needing to acquire the root identity, they are a very useful method to execute privileged tasks with the minimum necessary permissions. These facilities that the capabilities provide make them a very important security element in many systems. In this way, the capabilities are used in virtualization environments such as Linux or Docker containers, where they play a fundamental role in the management of security contexts.

The main advantage is to avoid granting a process to raise privileges at the superuser level when you really do not need more than certain permissions for a specific operation. In this table we can see some Linux capabilities with a description:

Figure 4.7: Linux capabilities

The Linux kernel prefixes all capability constants with CAP_ prefix . For example, CAP_CHOWN makes changes in bits UIDs and GIDs to change the owner of a file.

Listing all capabilities

The Linux libcap packages incorporate commands and binaries for listing and managing capabilities, among which we can highlight:

getcap: Allows listing capabilities of a file.

setcap: Allows assigning deleting capabilities of a file.

getpcaps: Allows listing capabilities of a process.

capsh: Provides a command-line interface for testing and exploring capabilities.

We can get an idea of the capabilities enabled by default with the capsh command that provides a command-line interface testing and exploring capabilities:

CAPAUDITWRITE allows writing access to the audit log

CAPAUDITCONTROL allows configuring the Linux audit subsystem

CAPNETADMIN allows configuring the network

CAP_SETPCAP allows set process control capabilities

The following command will start a new container using Python image and list capabilities with capsh --print command:

```
$ docker run --rm -it python sh -c 'apk add -U libcap; capsh --print'
Unable to find image 'python:latest' locally
latest: Pulling from library/python
85b1f47fba49: Pull complete
ba6bd283713a: Pull complete
817c8cd48a09: Pull complete
47cc0ed96dc3: Pull complete
4a36819a59dc: Pull complete
db9a0221399f: Pull complete
7a511a7689b6: Pull complete
1223757f6914: Pull complete
Digest: sha256:59d8481f4b2d21f2ac6623e986b4e91fa704112df3e7d9ddbe7315d4
a153ef5
Status: Downloaded newer image for python:latest
sh: 1: apk: not found
Current: = cap_chown, cap_dac_override, cap_fowner, cap_fsetid, cap_kill, cap_setgid, cap_setuid, cap_setpcap, cap_net_bind_service, cap_net_raw, cap_sys_chroot, cap_mknod, cap_audit_write, cap_setfcap+eip
Bounding set =cap_chown, cap_dac_override, cap_fowner, cap_fsetid, cap_kill, cap_setgid, cap_setuid, cap_setpcap, cap_net_bind_service, cap_net_raw, cap_s
```

Figure 4.8: Listing capabilities in a Docker container

At this point, we have reviewed the different capabilities activated by default in a docker container.

Add and drop capabilities

To provide or remove Linux permissions to different containers, Docker provides the following commands, where we can apply add or remove privileges through flags:

```
$ docker run --cap-add={capability}
```

```
$ docker run --cap-drop={capability}
```

We could add a specific capability with the command:

```
$ docker run --rm -it --cap-add
```

```
$CAP alpine sh
```

To drop capabilities from the root account of a container, we can use:

```
$ docker run --rm -it --cap-drop
```

```
$CAP alpine sh
```

To drop all capabilities and then explicitly add individual capabilities to the root account of a container we can use:

```
$ docker run --rm -it --cap-drop ALL --cap-add
```

```
$CAP alpine sh
```

For example, we can delete the CHOWN capability inside a container and then try to add a user. The action of adding a user will fail because the operation it needs CAP_CHOWN capability.

In the following command, we can see the action of changing ownership of a file or directory inside a ubuntu container:

```
$ docker run --cap-add=ALL --cap-drop=CHOWN -ti ubuntu sh
```

When executing the previous command, we can see that the action of changing ownership of a file or directory will fail and it will show Operation not permitted message:

```
$ docker run --cap-add=ALL --cap-drop=CHOWN -ti ubuntu sh
# useradd test
useradd: failure while writing changes to /etc/shadow
# chown nobody /usr/share
chown: changing ownership of '/usr/share': Operation not permitted
```

Figure 4.9: Drop CHOWN capabilities in a Docker container

In the same way, we can start a new container based on the Python image and eliminate the CHOWN capability so that the owner of a file inside the counter cannot be changed. When executing it, we see how the command returns an error code because this container's root account has been removed and, therefore, cannot change the ownership of a file or directory.

In the following command we are deleting CHOWN capability inside python container:

```
$ docker run -it --cap-drop CHOWN python /bin/bash
```

In the following screenshot, we can see the result of executing the previous command:

```
$ docker run -it --cap-drop CHOWN python /bin/bash
Unable to find image 'python:latest' locally
latest: Pulling from library/python
85b1f47fba49: Pull complete
ba6bd283713a: Pull complete
817c8cd48a09: Pull complete
47cc0ed96dc3: Pull complete
4a36819a59dc: Pull complete
db9a0221399f: Pull complete
7a511a7689b6: Pull complete
1223757f6914: Pull complete
Digest: sha256:59d8481f4b2d21f2ac6623e986b4e91fa704112df3e7d9ddbe7315d4
a153ef5
Status: Downloaded newer image for python:latest
root@ee7444d32500:/# chown nobody /usr/share/ca-certificates/
chown: changing ownership of '/usr/share/ca-certificates/': Operation no
t permitted
```

Figure 4.10: Drop CHOWN capabilities in a python container

Docker containers start with a reduced capacity set. By default, Docker enables the following capabilities: *chown*, *dac_override*, *fowner*, *kill*, *setgid*, *setuid*, *setpcap*, *netbindservice*, *netraw*, *syschroot*, *mknod*, *setfcap*, and *auditwrite*.

We can also remove all the capabilities that are enabled in Docker by default and check that the container stops working. Here we start a bash shell without the capabilities that are enabled by default:

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE  
--cap-drop=FSETID --cap-drop=FOWNER --cap-drop=KILL --cap-drop=MKNOD  
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID  
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NETBINDSERVICE  
--cap-drop=SYSCHROOT --cap-drop=AUDITWRITE ubuntu /bin/bash
```

It is also recommended to drop the setuid and setgid capabilities from containers that will be running on your hosts. The Linux kernel is responsible for managing the UID and GID space, and it's kernel-level syscalls that are used to determine if requested privileges should be granted.

To drop the setuid or setgid capabilities when you start a Docker container, you will need executing the following instruction:

```
$ docker run -it --cap-drop SETGID --cap-drop SETUID python sh
```

In the following screenshot, we can see the result of executing the previous command:

```
$ docker run -it --cap-drop SETGID --cap-drop SETUID python sh  
# cat /proc/self/status  
Name: cat  
State: R (running)  
Tgid: 7  
Ngid: 0  
Pid: 7  
PPid: 1  
TracerPid: 0  
Uid: 0 0 0 0  
Gid: 0 0 0 0  
FDSize: 64  
Groups:  
NStgid: 7  
NSpid: 7  
NSpgid: 7  
NSsid: 1  
VmPeak: 6080 kB
```

Figure 4.11: Drop SETGID and SETUID capabilities in a Python container

If we try to get the capabilities inside the container, we can see that UID and GID bits are equals to 0.

In the following screenshot we can see the result for listing capabilities:

```
# capsh --print
Current: = cap_chown, cap_dac_override, cap_fowner, cap_fsetid, cap_kill, cap_setpcap, cap_net_bind_service, cap_net_raw, cap_sys_chroot, cap_mknod, cap_audit_write, cap_setfcap+eip
Bounding set =cap_chown, cap_dac_override, cap_fowner, cap_fsetid, cap_kill, cap_setpcap, cap_net_bind_service, cap_net_raw, cap_sys_chroot, cap_mknod, cap_audit_write, cap_setfcap
Securebits: 00/0x0/1'b0
secure-noroot: no (unlocked)
secure-no-suid-fixup: no (unlocked)
secure-keep-caps: no (unlocked)
uid=0(root)
gid=0(root)
groups=
```

Figure 4.12: GID and UID bits after dropping these capabilities

At this point, we have reviewed that we can disable UID and GID bits to improve the security of our container and avoid a possible elevation of privileges.

Disabling ping in a container

For disabling ping in a python container, we can use the following command that drops the NET_RAW capability in the Python container. If we try the ping command inside the container, it will return the message ping: Lacking privilege for raw socket.

In the following command we are dropping net_raw capability inside python container:

```
$ docker run --it --cap-drop NET_RAW python sh
```

In the following screenshot, we can see the result of executing the previous command:

```
$ docker run -it --cap-drop NET_RAW python sh
Unable to find image 'python:latest' locally
latest: Pulling from library/python
85b1f47fba49: Pull complete
ba6bd283713a: Pull complete
817c8cd48a09: Pull complete
47cc0ed96dc3: Pull complete
4a36819a59dc: Pull complete
db9a0221399f: Pull complete
7a511a7689b6: Pull complete
1223757f6914: Pull complete
Digest: sha256:59d8481f4b2d21f2ac6623e986b4e91fa704112df3e7d9ddbe7315d4a
153ef5
Status: Downloaded newer image for python:latest
# ping 8.8.8.8
ping: Lacking privilege for raw socket.
```

Figure 4.13: Disable ping in Python container

In the previous example, we have disabled the use of RAW and PACKET sockets. In this way, using the capabilities, it is possible to get users or processes to perform privileged tasks with the level of granularity that we want.

The best practice at this point is to eliminate all capacities and add only those we need in our container with the flags - cap-drop and --cap-add . In this example, eliminating all the capabilities, we see how we cannot ping or change the host name.

In the following command, we can see the result for dropping all capabilities in alpine container. When trying to execute ping command, it will return permission denied:

```
$ docker run -ti --cap-drop=all alpine sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
e7c96db7181b: Pull complete
Digest: sha256:769fddc7cc2f0a1c35abb2f91432e8beecf83916c421420e6a6da9f8975464b6
Status: Downloaded newer image for alpine:latest
/ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: permission denied (are you root?)
/ # hostname foo
hostname: sethostname: Operation not permitted
```

Figure 4.14: Disabling all capabilities in alpine container

In this way, using Linux capabilities, we have been able to make users and processes perform tasks that require privileges with greater granularity. For example, by eliminating the setuid and setgid bits, a possible attacker who finds a vulnerability within the container, could not obtain an execution shell since it could not obtain a shell with root privileges.

Adding capability for managing network

In the next example, we are using --cap-add=NET_ADMIN to add capabilities to configure the network and networking control. By adding this capability, we can disable the network interface with the link set eth0down command, and when trying ping command, it would return network is unreachable.

In the following command we are adding net_admin capability inside python container:

```
$ docker run -ti --cap-add=NET_ADMIN python sh -c "ip link set eth0 down"
```

In the following screenshot, we can see the result of executing the previous command:

Figure 4.15: Enable capability for managing network

One of the capabilities is CAPNETRAW , and this option has several security implications related to the sending of packages since it allows any package to be generated, and impersonation attacks could be made to perform MITM attacks from a container. It is recommended to disable this capability along with those that are not necessary for the execution of a containerized application. For the rest of the capabilities, highlight the one of CAPNETADMIN that does not activate it by default, so we should untrust images that run this capability since it gives full control of networking.

In the following commands, we are checking to drop NET_RAW capability inside busybox container:

Figure 4.16: Disabling capability NET_RAW for managing network

At this point, we have reviewed how we can manage a network with capabilities *NETRAW* and *NETADMIN*.

Execution of privileged containers

There are times when you need your container to have special kernel capabilities that would normally be denied. This could include mounting a USB drive, modifying network settings, or creating a new Unix device. In the following code, we try to change the MAC address of our container in the eth0 interface:

```
$ docker run --rm -ti ubuntu /bin/bash
root@b328e3449da8:/# ip link ls
1: lo: mtu 65536 qdiscnoqueue state ...
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0: mtu 1500 qdiscnoqueue state ...
link/ether 02:42:0a:00:00:04 brd ff:ff:ff:ff:ff:ff
root@b328e3449da8:/# ip link set eth0 address 02:0a:03:0b:04:0c
```

RTNETLINK answers: Operation not permitted

When performing this operation, we see how it is not allowed since the Linux kernel blocks it in the container. However, assuming we need this functionality for our container to work, the easiest way to expand the privileges of a container is to launch it with the argument *--privileged = true* .

In the following command we are executing ubuntu container with full privileges:

```
$ docker run -ti --rm --privileged=true ubuntu /bin/bash
```

In the result of the command with this flag, we see how the MAC address has been modified correctly.

```
root@88d9d17dc13c:/# ip link ls
1: lo: mtu 65536 qdiscnoqueue state ...
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0: mtu 1500 qdiscnoqueue state ...
link/ether 02:42:0a:00:00:04 brd ff:ff:ff:ff:ff:ff
root@88d9d17dc13c:/# ip link set eth0 address 02:0a:03:0b:04:0c
```

```
root@88d9d17dc13c:/# ip link ls  
1: lo: mtu 65536 qdiscnoqueue state ...  
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
9: eth0: mtu 1500 qdiscnoqueue state ...  
link/ether 02:0a:03:0b:04:0c brdff:ff:ff:ff:ff:ff
```

With privileged access within the container, we provide more capabilities to perform operations normally performed by root. For example, let's try to create a device while mounting a disk image.

In the following command we are executing fedora container with full privileges for getting a bash shell:

```
$ docker run --privileged -i -t fedora /bin/bash
```

In the following screenshot, we can see the result of executing the previous command:

Figure 4.17: Disabling capability NET_RAW for managing network

The problem with using the `--privileged=true` flag is that it gives your container very wide privileges, and in most cases, you probably only need one or two kernel capabilities to perform the necessary operations. As we can see, the privileged container can access much more hardware than the non-privileged container.

Docker content trust

Docker Content Trust (DCT) is a solution offered by Docker to ensure that containers are not compromised from a security point of view, and their origin and traceability are maintained.

This Docker mechanism allows developers to sign their content, completing the reliable distribution mechanism. When a user downloads an image from a repository, this mechanism allows you to check the signatures of the images, receiving a certificate that includes the public key, which allows you to verify that the image comes from the one that made the upload to the docker registry.

If you are using the Docker hub, it is very simple to use DCT exporting the environment variable:

```
$ export DOCKERCONTENTTRUST=1
```

`DOCKERCONTENTTRUST` is a feature that restricts the Docker client from using only image tags that have been signed before performing a pull. To enable this option, which by default is disabled, we need to define the `DOCKERCONTENTTRUST` environment variable or run Docker engine with the option `--disable-content-trust = false`.

All keys are stored on the client-side, and only the timestamp and signature are stored as metadata along with the tags and image in the Docker log. If we download an image from a trusted repository, we see how it also checks the signature of the image.

In the following screenshot, we can see the result of pulling the Python Docker image with docker content trust enabled:

Figure 4.18: Downloading Python image with DCT

DCT has the ability to protect against some attack scenarios among which we can highlight:

Protection of malicious code in images: For example, if a possible attacker wants to make a modification to an official image to introduce malicious code, this mechanism protects you from it.

Protection against repeated attacks: Repeat attacks are those in which a malicious user tries to pass an earlier version of an application, which has been compromised, such as the latest legitimate version. The security mechanism of DCT will maintain the integrity of the image through the use of timestamps.

Protection against key commitments: If a key is compromised, this mechanism creates a new key and would become to create the image with this new key.

Signing images mechanism

Now we will review both the integrity of the images and the different layers that compose it. For each layer and for each image, they have a SHA256 hash that allows verifying if that image has been modified. However, if any of these images are downloaded through an insecure network or are simply published by a third party or malicious user, it is not possible only with the hash information to be able to detect it.

To do this, there are two commands that are entered in the Dockerfile that allow you to sign the image at the time of building or publishing them in the Docker record itself, and therefore, to be able to verify signature file when they are downloaded from the registry.

For the signing of images, we must configure environment variables DOCKERCONTENTTRUST and DOCKERCONTENTTRUST_SERVER.

The first environment variable allows you to enable and disable DCT verification. If enabled, the integrity of the image will be verified, relying on the official Docker notary server indicated in the second environment variable.

The second-order allows you to define the URL where the notary server is located. In most cases, companies feed on the images from the Docker hub image repository, where the notary service will be carried out by the Docker company.

Here are how to enter the previous environment variables in our Dockerfile to sign the image:

```
export DOCKERCONTENTTRUST=1  
export DOCKERCONTENTTRUST_SERVER="http://notary.docker.io"
```

When an image is uploaded, the Docker client will return a string that represents the image hash. This hash is with which the image will be validated when performing a pull.

In the following screenshot, we can see the result of pulling the Python Docker image with the hash value:

```

$ docker pull python@sha256:35ff9f44818f8850f1d318aa69c2e7ba61d85e3b93283078c10e56e7d864c183
sha256:35ff9f44818f8850f1d318aa69c2e7ba61d85e3b93283078c10e56e7d864c183: Pulling from library/python
c5e155d5a1d1: Already exists
221d80d00ae9: Already exists
4250b3117dca: Already exists
3b7ca19181b2: Already exists
425d7b2a5bcc: Already exists
dc3049ff3f44: Pull complete
472a6afc6332: Pull complete
5f79c90f8d7c: Pull complete
1051ee813012: Pull complete
Digest: sha256:35ff9f44818f8850f1d318aa69c2e7ba61d85e3b93283078c10e56e7d864c183
Status: Downloaded newer image for python@sha256:35ff9f44818f8850f1d318aa69c2e7ba61d85e3b93283078c10e56e7d864c183
docker.io/library/python@sha256:35ff9f44818f8850f1d318aa69c2e7ba61d85e3b93283078c10e56e7d864c183
[node1] (local) root@192.168.0.28 ~
$ docker pull python@sha256:35ff9f44818f8850f1d318aa69c2e7ba61d85e3b93283078c10e56e7d864c183
invalid checksum digest length

```

Figure 4.19: Downloading Python image verifying SHA256 hash

Each time an image is attempted to be downloaded; in this way, Docker will verify that the hash matches that of the original image. Any update of the image will result in the generation of a new hash. In this way, we avoid impersonation of images and ensure downloading an image in a secure way.

When enabling DCT, the Docker engine will only download those images that have been signed and will deny the execution of those images whose signatures do not match, in this case, the user will receive the error remote trust data does not exist for docker.io/

In the following screenshot, we can see the result of enabling and disabling docker content trust when downloading an image from the Docker hub:

```

$ export DOCKER_CONTENT_TRUST=1
[node1] (local) root@192.168.0.28 ~
$ docker pull jmortegac/linux_tweet_app:1.0
Error: remote trust data does not exist for docker.io/jmortegac/linux_tweet_app: notary.docker.io does not have trust
data for docker.io/jmortegac/linux_tweet_app
[node1] (local) root@192.168.0.28 ~
$ export DOCKER_CONTENT_TRUST=0
[node1] (local) root@192.168.0.28 ~
$ docker pull jmortegac/linux_tweet_app:1.0
1.0: Pulling from jmortegac/linux_tweet_app
bc95e04b23c0: Pull complete
a21d9ee25fc3: Pull complete
9bda7d5afdf39: Pull complete
e64d34b6ad71: Pull complete
a7e018a2b8ff: Pull complete
Digest: sha256:db9f2e75b91780c804c283081123fbelb2b4080fe124eeb040f8ad1bfd70fe38
Status: Downloaded newer image for jmortegac/linux_tweet_app:1.0
docker.io/jmortegac/linux_tweet_app:1.0

```

Figure 4.20: Downloading Docker image with DOCKERCONTENTTRUST

At this point, we have reviewed how we can manage the verification of Docker images with DCT.

Secure download in Dockerfiles

Regarding the secure download in Dockerfiles, in most cases, the providers add the signatures of verification. For example, the Dockerfile for the python image makes use of the GPG key to verify the signature of the package being downloaded.

In this screenshot we can see the use of GPG key for download packages in python docker image:

```

ENV GPG_KEY E3FF2839C048B25C084DEBE9B26995E310250568
ENV PYTHON_VERSION 3.8.0a4

RUN set -ex \
    \
    && wget -O python.tar.xz "https://www.python.org/ftp/python/${PYTHON_VERSION%*[a-z]}/Python-$PYTHON_VERSION.tar.xz" \
    && wget -O python.tar.asc "https://www.python.org/ftp/python/${PYTHON_VERSION%*[a-z]}/Python-$PYTHON_VERSION.tar.xz.asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --batch --keyserver ha.pool.sks-keyservers.net --recv-keys "$GPG_KEY" \
    && gpg --batch --verify python.tar.asc python.tar.xz \
    && { command -v gpgconf > /dev/null && gpgconf --kill all || :; } \
    && rm -rf "$GNUPGHOME" python.tar.asc \
    && mkdir -p /usr/src/python \
    && tar -xJC /usr/src/python --strip-components=1 -f python.tar.xz \
    && rm python.tar.xz \
    \

```

Figure 4.21: Download a package in a secure way from a Dockerfile

At this point, we have reviewed some best practices that can be applied in Dockerfiles for downloading files and packages in a secure way.

Notary as a tool for managing images

Docker notary is a tool that allows you to publish and manage images reliably. The use of this tool to sign your content ensures that those who want to download an image, the content is reliable.

Some of the notary's objectives include guaranteeing confidence in the images we download, either from a public or private repository, the delegation of trust between users, or the reliable distribution over repositories or channels that are not trusted.

Notary aims to make downloading images a safer and more reliable process by making it easier for people to publish and verify Docker images.

Notary consists of the server and client parts. The client part is installed on the local machine and handles the storage of the keys locally, as well as communication with the notary server. For more information on how to compile and configure the notary server, see the GitHub repository <https://github.com/docker/notary>.

You can find official Docker hub notary servers at <https://notary.docker.io>, and a precompiled notary binary for 64-bit Linux or macOS is available in the GitHub repository <https://github.com/theupdateframework/notary/releases>.

Docker registry

Docker provides a software distribution mechanism, also known as a registry, which facilitates the discovery and distribution of Docker images. The concept of registration is fundamental as it provides a set of utilities to package, send, store, discover, and reuse images. The best known Docker registry is the Docker hub.

What is a registry?

A Registry is one of the key pieces when creating our Docker environments as soon as we start creating our own images. An own registry will allow the different docker engines that we have to download the images that we develop and that we do not necessarily want to be public.

We also have the possibility to set up our own registry. Having the registry in our infrastructure saves us bandwidth and gives us better access / download time. This may seem unimportant today, in clusters where we release updates or self-healing comes into play has its importance.

The following image shows a workflow where a user builds an image and loads it into the registry:

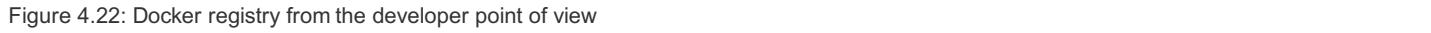


Figure 4.22: Docker registry from the developer point of view

The idea behind the Docker registry is that developers can extract the image from the registry to create other containers and deploy them either in the public cloud or in an organization's private servers.

Docker registry in Docker hub

Docker hub is the main image registry service. This service is offered as a Software as a Service platform with several usage plans.

The Docker registry works in a very similar way to Git. Each image, also known as a repository, is a succession of layers. In this way, every time we build our image locally, the Docker registry only stores the difference from the previous version, making the image creation and distribution process much more efficient.

To run and deploy a Docker registry on your own server, you can do it in several ways to store and distribute your own Docker images. For Linux distributions that include a Docker registration package (such as Fedora and Red Hat Enterprise Linux), you can install the package and start the service. For other distributions, you can run the image of the official Docker.io registry container to deploy the service.

Docker registry is an open-source project that can be installed on any server to create your own registry and upload your images privately. The goal is to have an alternative to the Docker hub to track the images hosted on your own server.

It allows deploying private repositories for internal use where to host images locally in a controlled environment. The official version of this service can be found in the Docker hub on the URL http://hub.docker.com/_/registry.

Creating Docker local registry

This image contains an implementation of the Docker Registry HTTP API V2 for use with Docker version >= 1.6.

Below are the steps to set up a private docker registry on your own server:

Once you have executed the next command, we have a container of the type registry running in the IP address of the Docker host.

In the following command we are executing registry container in detach mode with port 5000 exposed:

```
$ docker run -d -p 5000:5000 --restart = always --name registry registry:2
```

In the following screenshot, we can see the result of executing the previous command:

```

$ docker run -d -p 5000:5000 --restart=always --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
c87736221ed0: Pull complete
lcc8e0bb44df: Pull complete
54d33bcb37f5: Pull complete
e8afc091c171: Pull complete
b4541f6d3db6: Pull complete
Digest: sha256:77a8fb00c00b99568772a70f0863f6192ff2635e4af4e22e4d9c622edeb5f2de
Status: Downloaded newer image for registry:2
d47e0b90b502870403e3f634632ffd4012c792b02aba11f4264cc5eed56c7089
[node1] (local) root@192.168.0.28 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
d47e0b90b502        registry:2          "/entrypoint.sh /etc..."   2 minutes ago      Up 2 minutes       0.0.0.
0:5000->5000/tcp   registry

```

Figure 4.23: Downloading a Docker image for creating a local registry

The previous command starts the image of the registry available in Docker hub, exposes TCP port 5000 on the system so that clients outside the container can use it, and executes it as a foreground process. For testing this container, you can upload and download images in the private repository.

For example, you can download the hello-world image available in the Docker hub registry.

```
$ sudo docker run --name myhello hello-world
```

The next step is tagging the hello-world Docker image.

We can use the docker tag command to give a name to the docker image:

```
$ docker tag hello-world localhost:5000/hello-me:latest
```

In the following screenshot, we can see the result of executing the previous command:

Figure 4.24: Tagging hello-world Docker image

The next step is to push the image in the registry.

To save the hello-world image in the local Docker registry, we execute the command:

```
$ docker push localhost:5000/hello-me:latest
```

In the following screenshot, we can see the result of executing the previous command:

Figure 4.25: Pushing hello-world Docker image on the local registry

The next step is to ensure that you can recover the image from the registry. First, you need to delete the current image with the docker rm command, and then retrieve it from your local registry:

```
$ sudo docker rm myhello  
$ sudo docker rmi hello-world localhost:5000/hello-me:latest  
$ sudo docker pull localhost:5000/hello-me:latest
```

In the following screenshot, we can see the result of executing the previous commands:

Figure 4.26: Deleting hello-world Docker image and pulling from the local registry

Finally, we can verify that the image has been downloaded to our host Docker and we can execute the image.

```
$ docker images  
$ docker run -it localhost:5000/hello-me
```

In the following screenshot, we can see the result of executing the previous commands:



Figure 4.27: Checking Docker images and running Docker image from localhost

In summary, setting up a private Docker registry offers developers the ability to send and extract images without using the public Docker registry.

Conclusion

Docker containers present unique security challenges. There are some Docker security concerns to keep in mind. First, running containers and applications with Docker means running the Docker daemon, which requires root privileges. Other concerns include container flexibility, which makes it easy to run multiple instances of containers. Many of these containers can be at different levels of security patches.

Like any other technology, Docker is not exempt from possible security problems. To minimize them, it is best to apply good practices and audit our infrastructure frequently for vulnerabilities. In the next chapter, we will review tools for testing security in the Docker host.

Questions

Which is responsible for creating and managing containers, which includes creating file systems, assigning IP addresses, routing packets, process management, and many more tasks that require administrator privileges?

Which is a directory that is separate from the root file system of the container and is managed directly by the daemon docker process and can be shared between containers?

Which bits are special permissions that are used to access directories and files in the operating system by users that do not have root permissions?

Which environment variable verify the images you download from the Docker registry or Docker hub are trusted and signed?

Which tool allows us to manage what permissions have a process for accessing the kernel functions and allow segregating root user privileges?

CHAPTER 5

Docker Host Security

This chapter covers topics like AppArmor and Seccomp profiles, which provide kernel-enhancement features in order to limit system calls. Also, we will review tools such as Docker bench security, Lynis, and DockScan that follow security best practices in the Docker environment and some of the important recommendations that can be followed during auditing and Docker deployment in a production environment. The Docker Bench for Security can help test for docker content trust features and access control issues.

Analyzing the security of the host docker is also important since most of the attacks that can occur take advantage of a kernel vulnerability or because some package has not been updated. At this point, we will review some tools for auditing the security of the Docker Host.

Structure

Docker daemon security

Apparmor and Seccomp profiles

Docker bench security

Auditing Docker host with Lynis and DockScan

Objectives

Understanding Docker daemon security

Understanding Apparmor and Seccomp profiles

Knowing about Docker bench security

Knowing about auditing Docker host with Lynis and DockScan

Docker daemon security

The most important element of the Docker architecture is the Docker daemon process, which guarantees communication between containers and that traffic is protected by https.

When using the socket /var/run/docker.sock , it is important to check the access permissions by the users, in particular, that only the root user has to write permissions, and the Docker group does not contain users that can compromise the container.

At this point, it is important to note that Docker socket exposure can result in privilege escalation. Docker works primarily as a client that communicates with a daemon process called dockerd. This process with root privileges is a socket located in the path /var/run/docker.sock .

We can create new containers inside another container on the Docker host, for it the process /var/run/docker.sock must be mounted as a volume with the command:

```
$ docker run -it -v /var/run/docker.sock:/var/run/docker.sockdebian /bin/bash
```

Being the container based on a Debian image we can follow the installation guide, the commands to be executed would be:

<https://docs.docker.com/install/linux/docker-ce/debian/#install-using-the-repository>

Update packages and install packages that allow us to connect with repositories securely with HTTPS:

```
$ apt-get update
```

```
$ apt-get install apt-transport-https ca-certificates curl gnupg2 \
```

```
software-properties-common
```

In the following screenshot we can see the output of the previous command:

Figure 5.1: Updating system and installing packages

You can use the following command to configure the stable repository and update the local repository using apt-get update :

```
$ add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/debian \
$(lsb_release -cs) \
stable"
$ apt-get update
```

We then install the latest version of Docker CE:

```
$ apt-get install docker-ce
```

In the following screenshot we can see the output of the previous command:

Figure 5.2: Installing docker-ce package

We mount root from the host Docker with the commands:

```
$ docker run -it -v /:/host debian /bin/bash  
$ chroot /host  
$ /bin/bash
```

In the following screenshot we can see the output of the previous commands:

Figure 5.3: Mounting root inside Docker

In this way, we see how the Docker container starts a new mount point / in /host container. This is the host root file system, not the first container. The second container connects to /host , and you can check how effectively it is root on the host. In this way, we have proven that we have root access to the host from any process.

As we have seen, the Docker daemon runs with root permissions; therefore, it is very important to limit users who have control over the Docker daemon. We could give a series of recommendations on how they should configure access to the directories and files of the daemon to all users who are not responsible for the management.

Each of the files is indicated below, what permissions they should have in a default scenario (there is only one user with permissions to control):

At this point we have reviewed the security of Docker daemon and what are the default permissions for each service this process is using at low

level.

Auditing files and directories

As the Docker daemon runs with root privileges, all directories and files should be constantly audited to know all the activities and operations that are running. In order to correctly audit all events that take place on the host Docker, we can use the Linux audit daemon framework, which includes the following features:

Audit processes and file modification

Monitor system calls

Detect intrusions

Register commands by users

To correctly configure the audit daemon, it will be necessary to add new rules in the file `/etc/audit/rules.d/audit.rules`. Next, we will proceed to add the necessary rules to be able to audit the directories:



```
## Rules
-w /usr/bin/docker -k docker
-w /var/lib/docker -k docker
-w /etc/docker -k docker
-w /usr/lib/systemd/system/docker.service -k docker
-w /usr/lib/systemd/system/docker.socket -k docker
-w /etc/default/docker -k docker
-w /etc/docker/daemon.json -k docker
-w /usr/bin/docker-containerd -k docker
-w /usr/bin/docker.runc -k docker
```

Figure 5.4: Configuration audit rules

Once the rules have been added, it will be necessary to restart the audit daemon using the following command:

```
$ sudo service auditd restart
```

Finally, in order to review the logs generated during the audit, they can be found in the path `/var/log/audit/audit.log`.

Kernel Linux security and SELinux

At this point, we will introduce SELinux, which enables an additional layer of insulation. Security-Enhanced Linux (SELinux) is a Linux kernel security module that provides different security controls, among which we can highlight access controls, integrity controls, and role-based access control (RBAC). In addition, it provides privacy policies between the Docker host and containerized applications.

Red Hat or Fedora-based distributions have SELinux enabled by default, and the Docker daemon usually runs on a specific SELinux domain. In Debian-based distributions, AppArmor is enabled by default and provides a layer that works similarly.

SELinux is a tool created by the National Security Agency (NSA) of the United States to protect systems in general and subsequently integrated into the Linux Kernel. To determine if you have SELinux enabled in your distribution, you can run the `sestatus` command:

```
$ sestatus
```

SELinux status: enabled

SELinuxfs mount: /sys/fs/selinux

SELinux root directory: /etc/selinux

Loaded policy name: targeted

Current mode: permissive

Mode from config file: permissive

Policy MLS status: enabled

Policy deny_unknown status: allowed

Max kernel policy version: 28

For installation from Ubuntu repositories <https://packages.ubuntu.com/bionic/selinux> or Debian-based distributions we can do it through the command:

```
$ sudo apt-get install selinux
```

For example, on SELinux-enabled systems, it is possible to verify if the Docker Daemon runs safely using the -Z option on commands such as netstat and ps :

```
$ ps -efZ | grep docker
```

```
systemu:systemr:docker_t:SystemLow root 1873 1 2 07:21 ? 00:00:00
```

```
/usr/bin/docker -d --selinux-enabled
```

The output of the previous command shows that the Docker daemon is running in a secure way, as shown by the security context of the first column. The presence of docker_t keyword is a type of special domain with restricted rights. You can also see how the Docker daemon /usr/bin/docker contains --selinux-enabled parameter.

To start the Docker daemon with this property enabled, we can execute the following command.

```
$ docker -d --selinux-enabled
```

We can also verify if SELinux is enabled through the docker inspect command.

```
$ docker ps -q | xargs docker inspect --format '{{.Id}}: SecurityOpt = {{.HostConfig.SecurityOpt}}'
```

To use SELinux you must also have the relevant SELinux policy creation tools available. If you have a distribution with the yum package manager, for example, you can run yum -y install selinux-policy-devel .

Mandatory access control (MAC) tools in Linux impose security rules that ensure that not only the normal read-write-execute rules apply to files and processes, but more precise rules can be applied to processes at the kernel level.

For example, a MySQL process can only afford to write files under specific directories, such as /var/lib/mysql . The equivalent standard for Debian-based systems is AppArmor.

Apparmor and Seccomp profiles

AppArmor enables the administrator to assign each running program a secure profile, define file system access, network capacities, rules of execution, etc. Also, it provides protection for external and internal threats, enabling system administrators to associate a secure profile with each application that restricts that application's capabilities.

AppArmor allows you to limit what an application can do through a set of rules; specifically, these are the main features:

Limit certain users or processes to access a Daemon application or process.

Limit an application or Daemon to access certain processes.

Limit the resources that the application can access.

You can find more information in the official documentation:

http://wiki.apparmor.net/index.php/Main_Page

<http://en.wikipedia.org/wiki/AppArmor>

It is possible to check if Docker is using AppArmor through the docker info command in the security options section.

```
$ docker info
```

In the following screenshot we can see the output of the previous command:

Figure 5.5: Execution of docker info

With the docker info command also we can see information about the use of CPU, memory, and other information related to the kernel, operating system, and the directory where Docker is installed.

Installing AppArmor on Ubuntu distributions

We can find the AppArmor-profiles package within the repository of the different versions of Ubuntu: <https://packages.ubuntu.com/bionic/apparmor-profiles>.

In the following screenshot we can see the version of the AppArmor package:

The screenshot shows the Ubuntu Software Center interface. At the top, there's a breadcrumb navigation: » Ubuntu » Packages » bionic (18.04LTS) » admin » apparmor-profiles. Below that, it says [Source: apparmor] [xenial][xenial-updates][bionic][bionic-updates][cosmic][disco][disco-updates][eoan]. The main title is "Package: apparmor-profiles (2.12-4ubuntu5.1) [security]". A description below it reads "experimental profiles for AppArmor security policies". Under "Other Packages Related to apparmor-profiles", there's a list with one item: "apparmor (>= 2.8.96~2535-0ubuntu1~) user-space parser utility for AppArmor". At the bottom left, there's a "Download apparmor-profiles" button. On the right, there's a sidebar titled "Links for apparmor-profiles" which contains a placeholder message "No screenshot available. Sorry." with a camera icon. Below that is another section titled "Ubuntu Resources:" with a single link: "Bug Reports".

Figure 5.6: PackageAppArmor-profiles for Ubuntu

For installation, we can execute the following command on Ubuntu terminal:

```
$ sudo apt-get install apparmor-profiles
```

Once you have installed AppArmor, there are some directories that are common:

/etc/apparmor/: This contains the files that configure the daemon.

/etc/apparmor.d/: This contains the ruleset files that limit the access that an application has to the rest of the system.

In the following screenshot we can see the folder structure of /etc/apparmor.d/ directory:

Figure 5.7: AppArmor files and directories

Applications commonly used to configure and customize AppArmor include:

/usr/sbin/aa-enforce: This enable a profile or set of rules.

/usr/sbin/aa-logprof: This enable registration for a profile. To use this command, you must first enable the profile with the aa-enforce command.

/usr/sbin/aa-complain: This enables the profile for registration.

/usr/sbin/aa-genprof: This generates custom profiles.

/usr/sbin/aa-notify: This returns users and processes that have been denied access to an application.

/usr/sbin/aa-status: This informs you about active profiles. AppArmor considers each profile active to create a policy for the system, also available as /usr/sbin/apparmor_status.

At this point, we have reviewed the installation of AppArmor and the folder structure for checking the default configuration.

AppArmor in practice

To check if AppArmor is enabled in the Linux kernel of the Docker host, we can do it with the following command:

```
$ docker info | grep apparmor
```

Security Options: apparmor

We can also use docker inspect to verify if the property is enabled:

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: AppArmorProfile={{ .AppArmorProfile}}'
```

By default, Docker uses the AppArmorDocker-default profile that is located in the /etc/apparmor.d/docker/ path. You can find more information about it in the official documentation: <https://docs.docker.com/engine/security/apparmor/#understand-the-policies>

AppArmorDocker-default profile

We can check the status of AppArmor in the Docker host and identify whether the Docker containers are running or not with an AppArmor profile. To do this, we can execute the apparmor_status command.

```
$ apparmor_status
```

apparmor module is loaded.

10 profiles are loaded.

10 profiles are in enforce mode.

/sbin/dhclient

/usr/bin/lxc-start

/usr/lib/NetworkManager/nm-dhcp-client.action

/usr/lib/NetworkManager/nm-dhcp-helper

/usr/lib/connman/scripts/dhclient-script

/usr/sbin/tcpdump

docker-default

lxc-container-default

lxc-container-default-with-mounting

lxc-container-default-with-nesting

0 profiles are in complain mode.

2 processes have profiles defined.

2 processes are in enforce mode.

/sbin/dhclient (610)

0 processes are in complain mode.

Keep in mind that Docker-default is now displayed in application mode procedures as well. The value in parentheses is the container process's PID, as seen from the Docker host's PID namespace.

Enforce mode profiles deny AppArmor-based activities. Recording activities in complain mode profiles that breached some of the safest environments of the profile but do not block functionality.

Run container without AppArmor profile

We have some options to run the container so that the AppArmor profile is disabled or instruct Docker to use the default profile:

To execute a container with AppArmor profile, use the flag: --security-opt apparmor = docker-default .

To execute a container without AppArmor profile, use the flag: --security-opt apparmor = unconfined .

To verify that the new container is not running with an AppArmor profile, we can execute the apparmor_status command.

```
$ apparmor_status
```

apparmor module is loaded.

1 processes are in enforce mode.

```
/sbin/dhclient (610)
```

0 processes are in complain mode.

0 processes are unconfined but have a profile defined.

When executing the command, we can see that in the processes that appear in the enforce section, there are no instances of the docker-default profile. This implies that not the apparmor docker-default profile is associated with the started container.

Defense in-depth

In-depth defense allows multiple lines of defense to work together to provide improved overall defensive capabilities from the security point of view.

With the following command we can start a Ubuntu container with Seccomp disabled by default:

```
$ docker container run --rm -it --cap-add SYS_ADMIN --security-opt seccomp=unconfined ubuntu sh
```

To verify that AppArmor is working we can try to create 2 directories and try to group them with the mount command and the bind option:

mkdir mydir1; mkdir mydir2; mount --bind mydir1 mydir2

```
mount: mount /mydir1 on /mydir2 failed: Permission denied
```

The operation returns permission denied because the AppArmor profile denied the operation. We can start a new container without an AppArmor profile and retry the same operation to confirm that the default AppArmor profile is the one that denied the operation.

```
$ docker container run --rm -it --cap-add SYS_ADMIN --security-opt seccomp=unconfined --security-opt apparmor=unconfined ubuntu sh
```

mkdir dir1; mkdir dir2; mount --bind dir1 dir2

ls -l

In the following screenshot we can see the output of the previous commands:

Figure 5.8: Checking container without AppArmor profile

We can see the procedure was effective in the previous image. This demonstrates that the procedure in the past scenario was avoided by the default AppArmor profile.

Run container with Seccomp profile

Seccomp is an isolated space installation in the Linux Kernel that acts as a firewall that allows you to limit system calls (syscalls). We can check if Seccomp is enabled with the docker info command:

```
$ docker info | grep seccomp
```

We can create the following file that allows us to define the system calls that we want to block. In this example, we are blocking mkdir, chmod , and chownsys calls.

```
touch policy.json
```

```
{
  "defaultAction": "SCMPACTALLOW",
  "syscalls": [
    {
      "name": "mkdir",
      "action": "SCMPACTERRNO"
    },
    {
      "name": "chmod",
      "action": "SCMPACTERRNO"
    },
    {
      "name": "chown",
      "action": "SCMPACTERRNO"
    }
  ]
}
```

```
}
```

```
]
```

```
}
```

Then, we can execute the container based on the alpine distribution, passing the policy.json policy file as a parameter.

```
$ docker run --rm -it --security-opt seccomp:policy.json alpine sh
```

We can verify that the mkdir, chmod and chown commands cannot be used inside the container and when executing them we get the error Operation not permitted :

```
$ mkdirnewdir
```

```
$ chownroot:root bin
```

```
$ chmod +x /etc/resolv.conf
```

In the following screenshot we can see the output of the previous commands:

Figure 5.9: Checking container with Seccomp profile

At this point, we have reviewed the execution of Seccomp profile over an alpine container using the policy defined in JSON file configuration.

Reducing the container attack surface

Reducing the attack surface is a fundamental principle of security. Beyond securing the host, the container's underlying shared kernel architecture needs attention, as well as retaining general standard settings and particular container profiles.

For example, container security depends on the Kernel and Docker daemon that is accessed through system calls. At this point, although Docker has made significant improvements in the ability to call Seccomp profiles, these profiles only disable certain calls by default, but there are others that are available, leaving a large number of syscalls that can be invoked without any restriction.

Another example is the ability to link the Docker daemon process with the Unix Docker access group or the TCP port that allows containers to communicate with each other.

The ultimate goal in security is to obtain a balance between the insulation of the containers, and the communication needs between them. This implies taking measures both to limit the number of containers that are accessible to groups and to control the degree to which the containers interact with each other.

Docker bench security

Docker bench security is a useful tool to test the security of your Docker containers. The objective is to perform the Docker CIS checks against a container, and a report is generated that tells you if that container is potentially insecure at the level of permissions and access to resources.

The tool mainly focuses on best practices in areas such as file permissions and registry settings. The following links are the Docker CIS benchmark guides, which allow securing machines running Docker.

<https://www.cisecurity.org/cis-benchmarks>

<https://www.cisecurity.org/benchmark/docker>

<https://docs.docker.com/v17.09/compliance/cis>

These guides allow defining a series of guidelines to be followed by all members of a DevOps team. In this way, the audit and internal security teams would be aware of these guidelines in order to perform the corresponding compliance and security tests. The use of these guides and best practices help to reduce the margin of error when starting to work with an established base.

Docker bench security is a shell script that looks for common best practice patterns around the implementation of Docker containers in production. It is a set of bash scripts, which must be run as a root user on any machine with Docker installed, and the tool will produce a report with all the checks. From the point of view of the Docker host and Docker daemon settings, this is the best tool you can use to check these best practices.

The source code available in the following GitHub repository can be accessed: <https://github.com/docker/docker-bench-security>.

Docker bench runs in a container with high privileges and runs a set of tests against all the containers that are in the host Docker. The tests return the PASS, WARN, or INFO status, and we can see if, for each test, it passes the specific functionality.

The tool will inspect the following components:

Host configuration

The Daemon docker configuration

Docker daemon configuration files

Image container and compilation files

Runtime container

Docker security operations

To execute the tool, we can do it through an image that we can find in the Docker Hub, copying the following command in our Docker host: <https://hub.docker.com/r/docker/docker-bench-security>.

In the following screenshot we can see the state of the image in Docker hub:

The screenshot shows the Docker Hub page for the 'Docker Bench for Security' image. On the left, there's a preview of the Dockerfile with comments about the tool's purpose and dependencies. To the right, there are three tabs: 'Docker Pull Command' (containing the command 'docker pull docker/docker-bench-security'), 'Owner' (listing 'docker' as the owner), and 'Source Repository' (listing 'Github' with the URL 'docker/docker-bench-security').

```
# -----
# Docker Bench For Security v1.3.3
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Community Edition Benchmark v1.1.0.
# -----
Initialization Fri Jul 14 09:18:42 UTC 2017

[INFO] 1 - Host Configuration
[WARN] 1.1 - Ensure a separate partition for containers has been created
[NOTE] 1.2 - Ensure the container host has been Hardened
[PASS] 1.3 - Ensure Docker is up to date
[INFO]   * Using 17.06.0 which is current
[INFO]   * Check with your operating system vendor for support and security maintenance for Docker
[INFO] 1.4 - Ensure only trusted users are allowed to control Docker daemon
[INFO]   * docker:x:992:vagrant
[WARN] 1.5 - Ensure auditing is configured for the Docker daemon
[WARN] 1.6 - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.7 - Ensure auditing is configured for Docker files and directories - /etc/docker
[WARN] 1.8 - Ensure auditing is configured for Docker files and directories - docker.service
[INFO] 1.9 - Ensure auditing is configured for Docker files and directories - docker.socket
[INFO]   * File not found
[INFO] 1.10 - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[INFO]   * File not found
[INFO] 1.11 - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json
```

Figure 5.10: Docker bench security in Docker hub

To start the Docker bench security for analyzing the Docker host with a default configuration, we can execute the following command:

```
$ docker run -it --net host --pid host --cap-add audit_control \
-v /var/lib:/var/lib \
```

```
-v /var/run/docker.sock:/var/run/docker.sock \
-v /usr/lib/systemd:/usr/lib/systemd \
-v /etc:/etc --label dockerbenchsecurity \
docker/docker-bench-security
```

In the following screenshot we can see the output of the previous command:

Figure 5.11: Executing Docker bench security

We also have the possibility to run the bash script that we find in the GitHub repository. <https://github.com/docker/docker-bench-security/blob/master/docker-bench-security.sh>

Among the checks carried out by Docker bench, we can highlight:

Host configuration: This section checks the security over the host Docker configuration.

Daemon Docker configuration: This section offers recommendations about the security of the docker daemon. Everything in this section affects the configuration of the Docker Daemon as well as each running container.

Docker daemon configuration files: This section shows information about the configuration files used by the daemon Docker. This ranges from permissions to properties. Sometimes, these areas may contain information that you do not want others to know, which could be in a plain text format.

Daemon Docker configuration: This section shows information about the Docker daemon configuration and has the capacity for detecting containers that are running on the same Docker host and checking the access to each other's network traffic. By default, all containers that run on the same Docker host have access to each other's network traffic.

Also, it has the capacity for detecting if the Docker container is running as a root user. In this case, it shows a recommendation to create a user for the container.

Also, it shows a warning message if you are adding many capabilities that the Docker container is not using (overprivileged).

Below are some examples of warning messages that the report shows and recommendations for action for each message.

Docker daemon settings

[WARN] 2.2 - Restrict network traffic between containers

By default, all containers that run on the same Docker host have access to each other's network traffic. To avoid this, you must add the flag `--icc = false` in the process of starting the Docker daemon.

Docker daemon configuration files

[WARN] 4.1 - Create a user for the container

[WARN] * Running as root:

At this point, it is recommended to create another user other than root to run your containers.

[WARN] 5.3 - Verify that containers are running only a single main process

[WARN] * Too many processes running:

It is recommended to run only one process per container:

[WARN] 5.4 - Restrict Linux Kernel Capabilities within containers

[WARN] * Capabilities added: CapAdd = [audit_control]

You can use the --cap-drop = {} flag from the Docker run command to remove the additional capabilities of a container. Docker also supports adding and removing capabilities; therefore, it is recommended to remove all capabilities, except those that you intend to use inside the container.

[WARN] 5.13 - Mount container's root filesystem as read-only

[WARN] * Container running with root FS mounted R / W:

Its use of the flag --read-only is recommended. Using read-only containers helps ensure that data is not modified or manipulated.

Thanks to this tool and the generated report, we have access to almost 100 security recommendations to always keep in mind when we are going to use Docker in production. In the previous table, we can see the suggested safety recommendations in the form of a checklist.

Execution examples with Docker bench security

We see how at the host configuration level it performs the following checks:

Check that a separate partition for containers has been created.

Check that Docker is up to date.

Verify that only trusted users can control Docker daemon; what it does is check users belonging to the Docker group.

In the following screenshot we can see the output of Docker bench security in the Host configuration section:

Figure 5.12: Checking host configuration with Docker bench security

The checks marked with a warning must be reviewed:

[WARN] 1.5 - Ensure auditing is configured for the Docker daemon

[WARN] 1.6 - Ensure auditing is configured for Docker files and directories - /var/lib/docker

[WARN] 1.7 - Ensure auditing is configured for Docker files and directories - /etc/docker

[WARN] 1.11 - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json

In the following screenshot we can see the output of Docker bench security in the Docker daemon configuration section:

Figure 5.13: Checking Docker daemon configuration with Docker bench security

Docker daemon configuration files are related to the permissions of the files related to the Docker daemon, such as docker.service, docker.socket . What it does is verify that these files can only be run with root permissions.

In the following screenshot we can see the output of Docker bench security in the Docker daemon configuration files subsection:

Figure 5.14: Checking Docker daemon configuration files

In the following screenshot we can see the output of Docker bench security in the Container Images and Build Files subsection:

Figure 5.15: Checking Container Images and Build files

To solve the warning, you would have to execute the command:

```
$ export DOCKERCONTENTTRUST=1
```

If we execute a container, we see the warnings that it detects by default.

```
$ docker container run --detach -ti --name mypython python /bin/bash
```

In the following screenshot, we can see the output of Docker bench security when enabling Docker content Trust, and we execute the previous container:

Figure 5.16: Default warnings when running container

To solve the most critical warnings, we could re-execute the container, but limiting resources at the memory level, CPU, read-only permissions, and use a non-root user.

```
$ docker container run --detach -ti -u 1000 --read-only -m 256mb --securityopt=
```

```
no-new-privileges --cpu-shares=500 --pids-limit=1 --name mypython python
```

```
/bin/bash
```

At this point, we have reviewed the execution of the Docker bench security tool for checking the security configuration in the Docker host, showing the output of the report in specific sections.

Docker bench security source code

In this section, we are going to review some parts of the Docker bench security source code.

In GitHub repository are available the bash scripts available in the tests folder. <https://github.com/docker/docker-bench-security/tree/master/tests>

In the following screenshot we can see the scripts in GitHub repository:

Branch: master	docker-bench-security / tests /	Create new file	Upload files	Find file	History
 konstruktoid	use only year and month for version check #309 ...				✓ Latest commit 326e31f on 13 Apr
..					
1_host_configuration.sh	use mountpoint and DockerRootDir #332				10 months ago
2_docker_daemon_configuration.sh	use only year and month for version check #309				4 months ago
3_docker_daemon_configuration_files.sh	linting				10 months ago
4_container_images.sh	revert grep thought fail				5 months ago
5_container_runtime.sh	linting				10 months ago
6_docker_security_operations.sh	Improve docker-bench-security json output				10 months ago
7_docker_swarm_configuration.sh	fix test 7.4 using 5.25 as a model				9 months ago
99_community_checks.sh	Improve docker-bench-security json output				10 months ago

Figure 5.17: GitHub Repository

Next, we will review in more detail some scripts individually: `5containerruntime.sh`. This script allows you to check AppArmor is enabled in the container. In this case is using the instruction policy `=$(docker inspect --format 'AppArmorProfile={{ .AppArmorProfile }}' "$c")`

In the following screenshot we can see partial code of the previous script:

```
desc_5_1="Ensure AppArmor Profile is Enabled"
check_5_1="$id_5_1 - $desc_5_1"
starttestjson "$id_5_1" "$desc_5_1"

totalChecks=$((totalChecks + 1))

fail=0
no_apparmor_containers=""
for c in $containers; do
    policy=$(docker inspect --format 'AppArmorProfile={{ .AppArmorProfile }}' "$c")

    if [ "$policy" = "AppArmorProfile=" ] || [ "$policy" = "AppArmorProfile=[]" ] || [ "$policy" = "AppArmorProfile=<no value>" ]; then
        # If it's the first container, fail the test
        if [ $fail -eq 0 ]; then
            warn "$check_5_1"
            warn "* No AppArmorProfile Found: $c"
            no_apparmor_containers="$no_apparmor_containers $c"
            fail=1
        else
            warn "* No AppArmorProfile Found: $c"
            no_apparmor_containers="$no_apparmor_containers $c"
        fi
    fi
done
```

Figure 5.18: Script 5containerruntime.sh that checks AppArmor profile

The script also allows you to check if SELinux is enabled in the container, using the Docker inspect command using the SecurityOpt filter.

```
policy=$(docker inspect --format 'SecurityOpt={{ .HostConfig.SecurityOpt }}' "$c")
```

In the following screenshot we can see code for checking SecurityOpt filter:

```
desc_5_2="Ensure SELinux security options are set, if applicable"
check_5_2="$id_5_2 - $desc_5_2"
starttestjson "$id_5_2" "$desc_5_2"

totalchecks=$((totalChecks + 1))

fail=0
no_securityoptions_containers=""
for c in $containers; do
    policy=$(docker inspect --format 'SecurityOpt={{ .HostConfig.SecurityOpt }}' "$c")

    if [ "$policy" = "SecurityOpt=" ] || [ "$policy" = "SecurityOpt=[]" ] || [ "$policy" = "SecurityOpt=<no value>" ]; then
        # If it's the first container, fail the test
        if [ $fail -eq 0 ]; then
            warn "$check_5_2"
            warn "* No SecurityOptions Found: $c"
            no_securityoptions_containers="$no_securityoptions_containers $c"
            fail=1
        else
            warn "* No SecurityOptions Found: $c"
            no_securityoptions_containers="$no_securityoptions_containers $c"
        fi
    fi
done
```

Figure 5.19: Script 5containerruntime.sh checking SecurityOpt filter

4containerimages.sh: The script that allows you to check if a specific user has been created for the build images or the root user is being used.

```
user=$(docker inspect --format 'User={{.Config.User}}' "$c")
```

In the following screenshot we can see code for checking the user used for building images:

```
for c in $containers; do
    user=$(docker inspect --format 'User={{.Config.User}}' "$c")

    if [ "$user" = "User=0" ] || [ "$user" = "User=root" ] || [ "$user" = "User=" ] || [ "$user" = "User=[]" ] || [ "$user" = "User=<no v
    # If it's the first container, fail the test
    if [ $fail -eq 0 ]; then
        warn "$check_4_1"
        warn "* Running as root: $c"
        root_containers="$root_containers $c"
        fail=1
    else
        warn "* Running as root: $c"
        root_containers="$root_containers $c"
    fi
done
```

Figure 5.20: Script 4containerimages.sh checking user is being used inside the container

2dockerauthentication.sh . The script that allows you to carry out checks related to the configuration of the Docker daemon. For example, check if TLS authentication has been configured.

In the following screenshot we can see code for checking TLS authentication is configured for Docker daemon:

```

desc_2_6="Ensure TLS authentication for Docker daemon is configured"
check_2_6="$id_2_6 - $desc_2_6"
starttestjson "$id_2_6" "$desc_2_6"

totalChecks=$((totalChecks + 1))
if [ grep -i 'tcp:///' "$CONFIG_FILE" 2>/dev/null 1>&2 ] || \
[ $(get_docker_cumulative_command_line_args '-H' | grep -vE '(unix|fd)://') >/dev/null 2>&1 ]; then
if [ $(get_docker_configuration_file_args '"tlsverify":' | grep 'true') ] || \
[ $(get_docker_cumulative_command_line_args '--tlsverify' | grep 'tlsverify') >/dev/null 2>&1 ]; then
pass "$check_2_6"
resulttestjson "PASS"
currentScore=$((currentScore + 1))
elif [ $(get_docker_configuration_file_args '"tls":' | grep 'true') ] || \
[ $(get_docker_cumulative_command_line_args '--tls' | grep 'tls$') >/dev/null 2>&1 ]; then
warn "$check_2_6"
warn "* Docker daemon currently listening on TCP with TLS, but no verification"
resulttestjson "WARN" "Docker daemon currently listening on TCP with TLS, but no verification"
currentScore=$((currentScore - 1))

```

Figure 5.21: Script 2dockerdaemon_configuration.sh checking TLSauthentication

At this point we have reviewed some code scripts of the Docker bench security tool for checking the security configuration in the Docker host.

Auditing Docker host with Lynis and Dockscan

Lynis is an open-source security audit tool for evaluating the security of Linux and UNIX-based systems. Lynis execute directly on the Docker host so that it has access to the Linux kernel. We can find the source code and the installation in the following repositories:

<https://cisofy.com/lynis>

<https://github.com/CISOfy/Lynis>

<https://cisofy.com/documentation/lynis/get-started/#installation-manual>

Once installed, the audit system command performs the following checks:

Check the operating system

Perform a search for available tools and utilities

Checking any Lynis update

Perform tests with the enabled add-ons

Perform safety tests by category

Security scan status report

In the following screenshot we can see the options of Lynis command:

Figure 5.22: Lynis command options

First configurations are checking boot, services, and kernel:

Figure 5.23: Checking boot, services, and kernel

Second, it checks configurations related to users, groups, and authentication:

Figure 5.24: Checking users, groups, and authentication

Third, it checks configurations related to shells and file systems:

Figure 5.25: Checking configurations related to shells and filesystems

Fourth, it checks configurations related to storage and name services:

Figure 5.26: Checking configurations related to storage and name services

Fifth, it checks configurations related to networking:

Figure 5.27: Checking configurations related to networking

Last, it checks configurations related to Docker containers and security frameworks like AppArmor and SELinux:

Figure 5.28: Checking configurations related to Docker containers and security frameworks

At this point we have reviewed the execution of the Lynisecurity tool for checking the security configuration in the Docker host.

Auditing a Dockerfile

This command initializes the tests related to Docker and will perform an analysis related to security in the Dockerfile.

```
$ lynis audit dockerfile
```

In the GitHub repository, we can find the script used to analyze the Dockerfile.

<https://github.com/CISOfy/lynis/blob/master/include/helperauditdockerfile> .

The first thing the script does is to obtain the type of image (Debian, Fedora, Ubuntu).

In the following screenshot we can see code for checking the type of the image:

```
FIND=$(grep "^\$FROM" ${AUDIT_FILE} | sed 's/ /:space:/g')
```

```
for I in ${FIND}; do
```

```
    IMAGE=$(echo ${I} | sed 's/:space:/ /g' | awk '{ if ($1=="FROM") { print $2 }}')
```

```
    TAG=$(echo ${IMAGE} | cut -d':' -f2)
```

```
    Display --indent 2 --text "Found image:" --result "${IMAGE}"
```



```
    IS_DEBIAN=$(echo ${IMAGE} | grep -i debian)
```

```
    IS_FEDORA=$(echo ${IMAGE} | grep -i fedora)
```

```
    IS_UBUNTU=$(echo ${IMAGE} | grep -i ubuntu)
```

```
    IS_ALPINE=$(echo ${IMAGE} | grep -i alpine)
```

```
    IS_LATEST=$(echo ${TAG} | grep -i latest)
```



```
    if [ -n "${IS_DEBIAN}" ]; then IMAGE="debian"; fi
```

```
    if [ -n "${IS_FEDORA}" ]; then IMAGE="fedora"; fi
```

```
    if [ -n "${IS_UBUNTU}" ]; then IMAGE="ubuntu"; fi
```

```
    if [ -n "${IS_ALPINE}" ]; then IMAGE="alpine"; fi
```

```
    if [ -n "${IS_LATEST}" ]; then
```

```
        ReportWarning "dockerfile" "latest TAG used. Specifying a targeted OS image and version is better for reproducible results."
```

```
    fi
```

Figure 5.29: Checking configurations related to Docker containers and security frameworks

Next, check if a user has been found in the Dockerfile or if the root user is being used instead.

```

FIND=$(grep "USER" ${AUDIT_FILE} | cut -d' ' -f2 )
if [ -z "${FIND}" ]; then
ReportWarning "dockerfile" "No user declared in Dockerfile. Container will execute command as root."
else
USER=$(echo ${FIND})
Display --indent 2 --text "User" --result "${USER}"
Fi

```

Finally, the script is checking packages that are being downloaded using curl or wget and if the download is being done securely using HTTPS.

In the following screenshot we can see code for checking download packages:

```

if [ ${FILE_DOWNLOAD} -eq 1 ]; then
SSL_USED_FIND=$(egrep "(https)" ${AUDIT_FILE})

if HasData "${SSL_USED_FIND}"; then
SSL_USED="YES"
COLOR="GREEN"
else
SSL_USED="NO"
COLOR="RED"
ReportSuggestion "Use SSL downloads when possible to increase security (DNSSEC, HTTPS, validation of domain, avoid MitM)"
fi
Display --indent 2 --text "Integrity testing performed" --result "${SSL_USED}" --color ${COLOR}
HASHING_USED=$(egrep "(sha1sum|sha256sum|sha512sum)" ${AUDIT_FILE})
Display --indent 2 --text "Hashing" --result "${HASHING_USED}"
KEYS_USED=$(egrep "(apt-key adv)" ${AUDIT_FILE}| sed 's/RUN apt-key adv//g'| sed 's/--keyserver/Key Server:/g' | sed 's/--recv/Key'
Display --indent 2 --text "Signing keys used" --result "${KEYS_USED}"
Display --indent 2 --text "All downloads properly checked" --result "?"
else
Display --indent 2 --text "No files seems to be downloaded in this Dockerfile"
fi

```

Figure 5.30: Checking download packages

At this point we have reviewed some code scripts of the Lynis tool for checking the security configuration in the Docker host.

Dockscan for scanning Docker installations for security issues and vulnerabilities

This tool developed in ruby language <https://github.com/kost/dockscan> requires installation on your machine Ruby 2.0 or higher and the ruby docker-api gem. The installation is very simple and is executed with the following command:

```
$ gem install dockscan
```

In the following screenshot we can see the output of dockscan installation:

```
root@d088a3c75010:/# gem install dockscan
Fetching: excon-0.59.0.gem (100%)
Successfully installed excon-0.59.0
Fetching: multi_json-1.12.2.gem (100%)
Successfully installed multi_json-1.12.2
Fetching: docker-api-1.34.0.gem (100%)
Successfully installed docker-api-1.34.0
Fetching: dockscan-0.1.2.gem (100%)
Successfully installed dockscan-0.1.2
Parsing documentation for excon-0.59.0
Installing ri documentation for excon-0.59.0
Parsing documentation for multi_json-1.12.2
Installing ri documentation for multi_json-1.12.2
Parsing documentation for docker-api-1.34.0
Installing ri documentation for docker-api-1.34.0
Parsing documentation for dockscan-0.1.2
Installing ri documentation for dockscan-0.1.2
Done installing documentation for excon, multi_json, docker-api, docks
onds
```

Figure 5.31: Installing dockscan

This tool is a script developed in ruby that analyzes the installation of the host docker and the execution of containers. The output of DockScan offers a report where we can see the limits of the configured resources, as well as to detect the case that the host Docker allows a container to transfer traffic directly to other containers.

To start the analysis, we execute the command:

```
$ dockscan unix:///var/run/docker.sock
```

In the following screenshot we can see the output of the previous command:

```
$ dockscan unix:///var/run/docker.sock
[WARNINg] MultiJson is using the default adapter (ok_json). We recommend loading a different JSON libra
ry to improve performance.
Dockscan Report

High
Running Experimental version of Docker.: It is recommended to replace Docker version with stable and pr
oduction ready one.

Medium
Docker running without defined limits: It is recommended to define docker limits.
Docker running with IPv4 forwarding enabled: It is recommended to disable IPv4 forwarding by default.
```

Figure 5.32: Analyzing Docker socket

In the output of the previous command, we see how the tool detects how Docker is running without limits at the memory or CPU level and with the forwarding of ipv4 packets between the host network and that of the containers in execution. The lack of control of these cases is marked as a medium risk vulnerability (medium), and running Docker with an experimental version marks it as high risk (high).

If we want a more extensive report, we indicate that generating a report in TXT format:

```
[root@dockerlab001 ~]# more myreport.txt
Dockscan Report

-[ Medium ]-
=Docker running with IPv4 forwarding enabled=
Description:
Docker daemon reports it is running daemon with IPv4 forwarding enabled.
This is not recommended for production as it forwards network packets without rules.
Output:
Docker daemon reports it is running with automatic IPv4 forwarding.
Solution:
It is recommended to disable IPv4 forwarding by default.

-[ Low ]-
=Container have higher number of changed files=
Description:
Container have high number of changed files which is not recommended practice.
This is not recommended for production as data can be lost. It can also mean successful break in attempt.
Output:
475bed6f82c3124e9602e5992a2d546e1e80aab08364184440a389893703505e (/traefik_proxy_1 ) with IP: has more than 5 file changes: 9
/etc
/etc/traefik
/etc/traefik/traefik.toml
/run
/run/secrets
/tmp
/var
/var/run
/var/run/docker.sock

Solution:
It is recommended to have minimal number of changed files inside container and do not store data inside container. It is recommended to use volumes.

=Docker registries are not mirrored=
Description:
Docker daemon reports it is running configuration without registry mirrors.
If you set up local mirror, your docker host does not have to go directly to internet if not needed.
Output:
Docker daemon reports it does not have mirror registries.
Offending registry indexes:
docker.io

Solution:
It is recommended to setup mirror registry.
```

Figure 5.33: Getting a report about security issues found in the Docker Host

At this point we have reviewed the DockScan tool for auditing the security of the Docker installation.

Conclusion

The host machine can be defined as the most important part of the Docker environment since where the entire infrastructure is supported and where the containers will be executed, which is why it is advisable to follow a set of good practices when we are going to configure the machine where they will be hosted, and they will execute the different containers.

The ultimate goal is to minimize the attack vectors that can be produced on the host Docker itself. If containers from different clients are running on the same host, breaking the security of one of those containers could leave them exposed to the other containers. Because all the containers that run on the same Docker host share the same execution kernel, it makes sense to spend time securing the core of it.

In the next chapter, we will review some open source tools such as Clair with quay.io repository and anchor for discovering vulnerabilities in Docker images.

Questions

Which Linux kernel security module provides security controls among which we can highlight access controls, integrity controls, and RBAC?

Which tool provides protection for external and internal threats, enabling system administrators to associate a secure profile with each application that restricts that application's capabilities?

Which tool allows us to test the security of your Docker containers and focuses on best practices in areas such as file permissions and registry settings?

Which open source security audit tool is used for evaluating the security of Linux and Unix-based systems?

Which tool is a script developed in ruby that analyzes the installation of the host docker and the execution of containers?

CHAPTER 6

Docker Image Security

In this chapter, we will review some open source tools such as Clair with quay.io repository and anchored for discovering vulnerabilities in Docker images. In addition to ensuring that your container is correctly designed from a security point of view, all image layers in a container must be free

of known vulnerabilities. This is done by software in the Docker repositories that have the capacity to realize a static analysis of images.

In this chapter, the reader will learn about analyzing the security of docker images and discover vulnerabilities through the study of a series of static analysis tools of the different layers that compose an image. As a result, developers will have the capacity to detect vulnerabilities in container applications before uploading them to production.

Structure

Docker hub repository

Docker security scanning

Open-source tools for vulnerability analysis

Clair scanner and quay.io repository

Analyzing Docker images with anchore engine and anchore cli

Objectives

Knowing about Docker hub repository

Understanding Docker security scanning

Knowing about open source tools for vulnerability analysis

Knowing about Clair scanner and quay.io repository

Knowing about anchore engine and anchore cli for vulnerability analysis

Docker hub repository

Docker Hub is a repository of images created by the community, in which any user can create their image and upload it in the repository to share it with the community.

Within this repository, there are 2 types of images, depending on their origin. First of all, we have the official images that are the ones that are maintained by the main suppliers such as Apache, Ngnix, MongoDB, Ubuntu, Alpine.

On the other hand, we can find the images that have been created by users, and therefore, they are images that have been customized and adapted according to their needs for the project. Most likely, it has been customized from an official image.

Docker security scanning

Docker Security Scanning is a service available in Docker Hub for private repositories that compares the contents of a container layer by layer, by inspecting the binary packages in that container against the Common Vulnerabilities and Exposures (CVE) database.

Alternative open-source tools are available, all of which function in the same way:

Scan the image, separate the layers, and build a comprehensive content manifest.

Get the different layers that make up the images at the package level.

Compare the images manifest's content with the CVE and NVD server list of known vulnerabilities.

This scanning tool's effectiveness depends on:

Static analysis depth and integrity: the scanner's ability to discover the image's inner layers and the nature of those layers.

Vulnerability feeds quality: indicates coverage and how much the vulnerability lists need to be updated.

For reducing false positives, you can use white lists.

In the next section, we are going to start with the Docker security scanning process that allows you to start a review process of images in Docker hub repositories.

The Docker security scanning process

Docker security scanning is the tool that integrates directly with the official docker hub repository and allows you to automatically review images found in public repositories and private records in Docker Hub or Docker Cloud for known vulnerabilities.

Docker maintains a service in charge of searching for known vulnerabilities and uses the CVE database <https://cve.mitre.org> to find in our images libraries, binaries, and exploits of the programs that run on them. This service is available for Docker hub private repositories, in Docker cloud and on-premise version, being in all cases a paid service, although there are third-party services that offer similar functionality for free.

More information in documentation: <https://docs.docker.com/datacenter/dtr/2.5/guides/admin/configure/set-up-vulnerability-scans>.

Periodically, Docker analyzes all the images uploaded to the Docker hub, and once the analysis is finished, it provides us with a result of the different vulnerabilities, as well as the level of criticality of these, in addition to informing the CVE of each of them.

In the following screenshot we can see the output of analyzing layers in a Docker image:

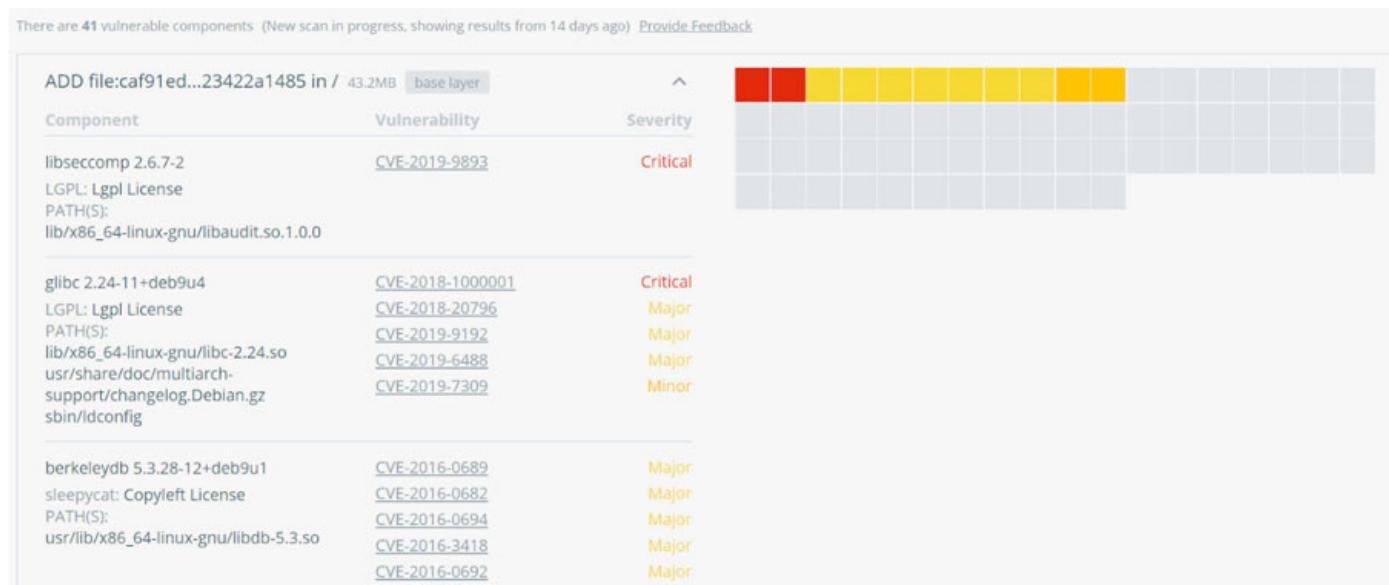


Figure 6.1: Analyzing layers in a Docker image

A single library or component can contain multiple vulnerabilities or exposures and Docker security scanning reports on each one. You can click an individual vulnerability report from the scan results and navigate to the specific CVE report data to learn more about it.

In the following screenshot we can see the information about specific CVE in a docker image:



Figure 6.2: Information about specific CVE in a Docker image

Vulnerabilities are given high, medium, or low-level criticality in the scan file. The criticality level is dependent on the score of the server assigned to the CVE code by the Common Vulnerability Score System (CVSS). They may be classified as follows, based on the score given to the vulnerability:

High: The vulnerability has a score within the range [8-10]

Medium: The vulnerability has a score within the range [4-7.9]

Low: Vulnerability has a score within the range [0.0-3.9]

When a new image is uploaded to the Docker Hub or Docker cloud, it launches a job that extracts that image from the docker registry and then sends that image to the scanning service that scans composite layers by analyzing each of the binaries with the CVE database.

Therefore, all the information related to the vulnerabilities found in that image is available. The next steps would be to identify false positives if you are really exposed to that vulnerability, and if so, correct it before performing the container deployment.

In conclusion, it is a good tool that Docker provides to know a little about the state of health in terms of security of official images and have knowledge if changes are being applied and patching those vulnerabilities.

Open-source tools for vulnerability analysis

The static vulnerability analysis is normally carried out by specialized tools for this function and is performed automatically, where the image created with its source code is analyzed, hence the static name, since the content of the This before its deployment in production.

This is a stage that currently, within the DevOps culture, is being carried out more and more, since it has been integrated into the entire project life cycle to be able to provide more and more information before the application reaches production.

This stage, if you enter the Docker world where they normally go hand in hand with the DevOps culture, the analysis of Docker image vulnerabilities lies in the first stage of building the solution. This analysis would be the only static analysis that would fall within the stages included in the continuous delivery and deployment pipelines and emerges as a new requirement at the security level due to the generation of Docker images as a packaging solution to be able to subsequently manage the execution of the application contained in these images as containers in different environments.

Mention that although at the Dockerfile level, you can also perform good practices, it is not until the time of image construction when you could verify which binary files and libraries have been included in the image itself. In addition, it is the immutability property of Docker images that makes static vulnerability analysis on a constructed image important since such image, as mentioned above, cannot be modified afterward, and it would be necessary to return to Modify the image using the Dockerfile file and rebuild the image.

In this section, we will review of the different open source tools or solutions that will be carried out to perform the static vulnerability analysis automatically.

Image scanners are an important set of tools. Tools such as Clair by CoreOS, Dagda, and Anchore can automatically check for image vulnerabilities, saving a great deal of time. When a vulnerability is detected, they can also send notifications via email and look for fixes.

Continuous integration with Docker

The scan can be easily integrated into continuous integration and delivery workflows so that scanning can be started automatically every time a developer completes a new container. Today, most administrators generally only discover a new vulnerability that is of high criticality by consulting the CVE database. The lower criticality vulnerabilities may not even be discovered. The scanner automates the process of identifying vulnerabilities that are not critical but could be exploited by potential attackers.

If a problem or vulnerability is discovered in the base image, and that image is reconstructed. Once the compilation is complete, the image is sent to the container platform log. The platform can detect that the image has changed. For constructions that depend on this image and have defined triggers, the platform allows you to automatically reconstruct the image of the application, incorporating libraries that have been changed during the reconstruction process.

Once the compilation is complete, the image is sent to the container platform log. The platform immediately detects changes in the image in its internal register, and, for applications that use it as a base image, the updated image is automatically implemented, ensuring that the code that is executed in production is identical to the most recent updated image. All these capabilities work together to integrate security capabilities into a process of continuous integration and continuous deployment (CI/CD) .

When a developer uploads the code to the source code control repository, it automatically displays the newly constructed image to test it. You may have to build the image and deploy (deploy) the new image automatically through the CI process.

Within the Docker ecosystem, the Dockerfile file describes how it will be built and what will be installed in the container so that the application can run on it. When running within a continuous integration environment, it will automatically generate and publish in the corporate Docker registry a container, including the software dependencies of that container.

A best practice to improve application security is to integrate automated security tests into your compilation or IC process. For example, we could integrate vulnerability scanner tools or dynamic application analysis:

Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools such as HP Fortify and IBM AppScan.

Tools such as Black Duck Hub and JFrog Xray that act as scanners that allow real-time verification of the code against a database of known vulnerabilities.

<https://www.blackducksoftware.com>

<https://jfrog.com/xray>

Below is a list of recommendations to ensure those platforms that are in the container construction environment. Within these recommendations, we can include Docker tools and others that allow you to automate and organize the source code.

Source code control: Source code control should be a common practice in DevOps security and operations equipment in order to ensure quality while contributing to the unit and integration testing. The main tools for source code control are GitHub <https://github.com> , GitLab <https://gitlab.com> , and Bitbucket <https://bitbucket.org> .

Compiler tools and controllers: Almost all the development teams use construction tools such as Bamboo and Jenkins. These platforms are an essential part of their automated compilation processes.

Security unit tests: Unit tests are an excellent way to execute test cases focused on specific functions and code modules. In addition, as the set of tests grows over time, it is essential to generate a set of regression tests to ensure that the vulnerabilities and security flaws have already been resolved.

Code analysis: It is advisable to implement code scans to verify that the code that is built is safe. Many tools have an integration through RESTful API within the development and continuous delivery cycle.

Component analysis: A useful method is to test libraries against the CVE list to decide if a vulnerable program is being used. Docker and some open source projects offer software that can be built into your security pipeline to test specific libraries against the CVE database.

In this section, we have reviewed the tools for working with continuous integration with Docker. In the next section, we will introduce some tools like CoreOS Clair and Dagda as a container vulnerability analysis services.

CoreOS Clair

Clair is a container vulnerability analysis service. It works through an API that analyzes each layer of the container looking for existing vulnerabilities in Debian, Ubuntu, and CentOS databases. It can also be used from the command line, as we see here. This tool has the capacity for reporting the list of known vulnerabilities that affect each container and notify users.

The methodology used for using this tool is by command line. Basically, it extracts all the layers of the image and notifies the vulnerabilities found and stores the information in a database. It also manages its own database that it updates from sources known as CVE.

<https://github.com/coreosclair>

<https://coreos.com/blog/vulnerability-analysis-for-containers/>

Clair is the security engine that uses the <https://quay.io> registry internally.

Dagda: the Docker security suite

Dagda is an open-source tool, developed in Python to perform static analysis of known vulnerabilities in Docker images/containers. It also helps you to monitor running Docker containers for detecting anomalous activities.

Dagda retrieves information about the software installed in your Docker image, such as the OS packages, library dependencies, modules, and matches it against a vulnerability database. This database is created by collating vulnerability data from sources such as NVD, SecurityFocus BID & Exploit-DB into a MongoDB database.

The database also stores your past static analysis scans performed on the Docker images and their result for a duration that you specify.

The project can be found in the following GitHub repository: <https://github.com/eliasgranderubio/dagda> .

We can find more information about the tool in the URL: <https://linuxsecurity.expert/tools/dagda> .

In the following image, you can see the Dagda GitHub repository:

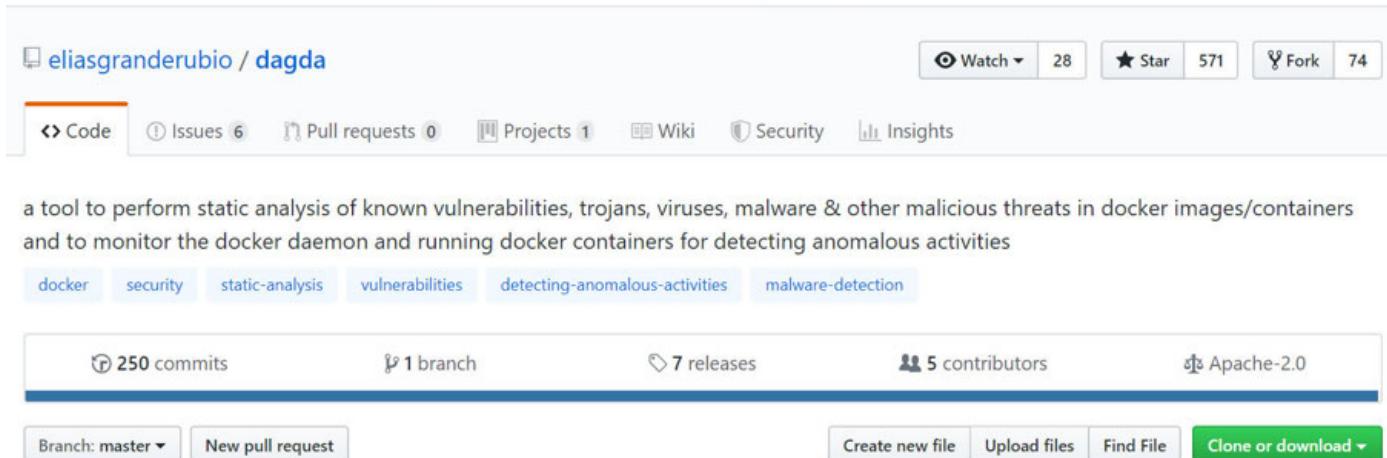


Figure 6.3: Dagda GitHub repository

Dagda supports multiple Docker base Linux images:

Red Hat/CentOS/Fedora

Debian/Ubuntu

OpenSUSE

Alpine Linux

Dagda uses OWASP dependency check and Retire.js to analyze packages and dependencies of various languages such as Java, Python, NodeJS, JS, Ruby, and PHP, identifying known vulnerabilities in Docker images.

It also runs ClamAV to search for malware or detect Trojanized images, and at the level of image monitoring, it integrates with Sysdig Falco as a tool to detect runtime anomalies and monitor containers on Unix environment. Falco is a tool that can be installed as an agent on each Docker host and internally functions analyzing system calls and kernel filters against the rules that are stored in a database for identifying attacks or anomalous calls inside the containers and in general in the Docker host.

If we follow the documentation that we can find in the GitHub repository, for its installation, we need to install the following modules on Python 3: Python 3.4.5, MongoDB 2.4, Pip3, PyMongo, Requests, Python-dateutil, Joblib, Docker-py, Flask, Flask-cors, PyYAML.

In the following image, you can see the Dagda architecture:

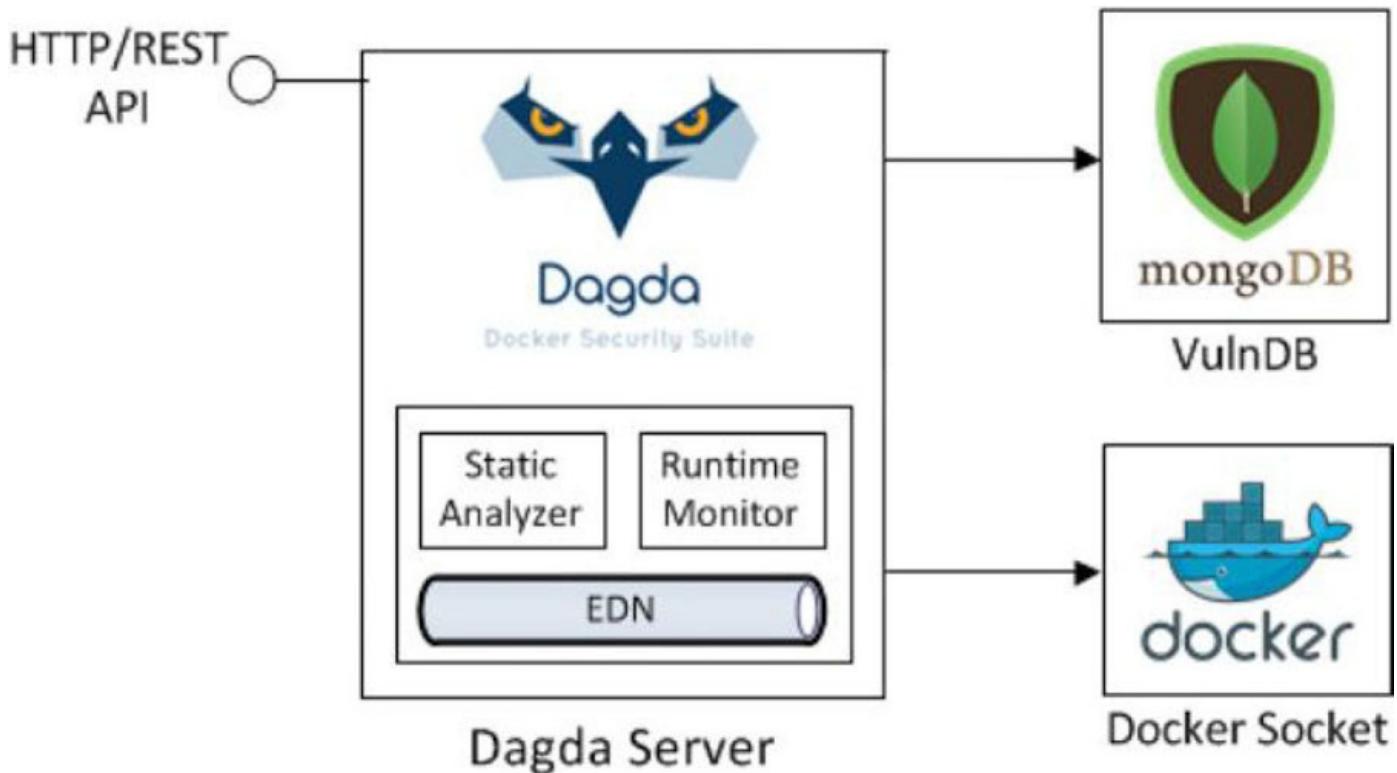


Figure 6.4: Dagda architecture

REST-API & Command-line interfaces are two ways with which you can interact with this Docker security suite. Every aspect of this tool can be controlled via the REST-API. You can have CLI access to this tool via the REST API. To interact with Docker containers, we can do it through the REST API. To execute it, we can do it with python3 to see what are the possible commands that we can execute.

In the following screenshot you can see the options of the Dagda Python script:

```

usage: usage: dagda.py [--version] [--help] <command> [args]

Dagda Commands:
check           perform the analysis of known vulnerabilities in
                docker images/containers
docker          list all docker images/containers
history         retrieve the analysis history for the docker images
monitor         perform the monitoring of anomalous activities in
                running docker containers
start           start the Dagda server
vuln            perform operations over your personal CVE, BID &
                ExploitDB database

Optional Arguments:
-h, --help       show this help message and exit
-v, --version    show the version message and exit

```

Figure 6.5: Dagda Python script options

The first thing to do is to start the Dagda server, and we can do it with the start options from the dagda.py Python script.

In the following screenshot you can see the options of the Dagda Python start-server script:

```
usage: dagda.py start [-h] [--server_host SERVER_HOST] [--server_port SERVER_PORT]
                      [--mongodb_host MONGODB_HOST] [--mongodb_port MONGODB_PORT]
                      [--falco_rules_file RULES_FILE]
```

The Dagda server.

Optional Arguments:

```
-h, --help            show this help message and exit
-s SERVER_HOST, --server_host SERVER_HOST
                     address/interface where the server binds itself. By
                     default, Dagda server binds to '127.0.0.1'
-p SERVER_PORT, --server_port SERVER_PORT
                     port where the server binds itself. By default, the
                     Dagda server port is 5000
-m MONGODB_HOST, --mongodb_host MONGODB_HOST
                     address/interface where the MongoDB is listening. By
                     default, MongoDB server is set to '127.0.0.1'
-mp MONGODB_PORT, --mongodb_port MONGODB_PORT
                     port where the MongoDB is listening. By default, the
                     MongoDB port is set to 27017
--falco_rules_file  sysdig/falco custom rules file (See 'Falco Rules' wiki
                     page [https://github.com/draios/falco/wiki/Falco-Rules]
                     for details)
```

Figure 6.6: Dagda Python start-server options

With the check option, we can perform a vulnerability analysis from a specific image, and with the history option, we can obtain the historical analysis made from the name of the image and the scan id.

```
$ python3 dagda.py check --docker_image
```

In the following screenshot, you can see the options for starting analysis in a Docker image:

```
usage: dagda.py check [-h] [-i DOCKER_IMAGE] [-c CONTAINER_ID]
```

Your personal docker security analyzer.

Optional Arguments:

```
-h, --help            show this help message and exit
-i DOCKER_IMAGE, --docker_image DOCKER_IMAGE
                     the input docker image name
-c CONTAINER_ID, --container_id CONTAINER_ID
                     the input docker container id
```

Figure 6.7: Dagda options for starting analysis in a Docker image

Successful acceptance returns a scanned ID, which you then use to retrieve the scan reports.

```
$ python3 dagda.py history --id
```

With the vuln option, we can initialize the vulnerability database and indicate if we want to filter by a specific CVE code. The first thing that should be done is to run the script with the -- init option to initialize the database with updated information about database vulnerabilities such as CVE, exploit database, and Red Hat security advisories.

In the following screenshot, you can see the options for checking vulnerabilities in an image:

Figure 6.8: Dagda options for checking vulnerabilities in an image

At this point, we have reviewed the Dagda script for checking vulnerabilities in Docker images.

OWASP dependency check

OWASP dependency check is an analysis tool that allows you to scan the images layer by layer, allowing you to analyze several languages such as Java, Python, Node.js, JavaScript, Ruby, and PHP. In the case of JavaScript code and Node.js, use the library Retire.js.

<https://www.owasp.org/index.php/OWASPDependencyCheck>

Internally, it performs a scan and collection of information about the pom.xml and manifest files in the case of Java projects and JAR files. In the case of projects with JavaScript, the target is to analyze the package.json file and the NPM dependencies. This information is compared with the NVD and CVE database.

The official project page can be found within the OWASP project and can be installed as a command-line tool or as a maven plugin to integrate it into projects as if it were a library.

If we analyze a project, we have the possibility of generating a report with the vulnerabilities detected. We can see an example report at <https://jeremylong.github.io/DependencyCheck/general/SampleReport.html>.

The tool is also available as a Docker image in the public Docker hub repository https://hub.docker.com/r/deepfenceio/deepfence_depcheck.

```
$ docker pull deepfenceio/deepfence_depcheck
```

In the following screenshot we can see the output of the previous command:

Figure 6.9: Downloading image form Docker hub

We can see the options and commands offered if we check the Docker image with the -h parameter:

```
$ docker run -ti -v /var/run/docker.sock:/var/run/docker.sock -v  
/var/lib/docker:/fenced/mnt/host/var/lib/docker/:rw -v  
/:/fenced/mnt/host/:ro -v /home/sandman/db:/tmp:rw  
deepfenceio/deepfence_depcheck -h  
usage: /usr/local/bin/start_services.sh options
```

In the following screenshot we can see the output of the previous command:

Figure 6.10: Options for deepfence Docker image

Also, we can see execution examples for deepfence Docker image:

Figure 6.11: Execution examples for deepfence Docker image

The first step before analyzing our images is to build the initial vulnerability database. The following command will initialize the database with data recovered from public database vulnerability

```
$ docker run -ti -v /var/run/docker.sock:/var/run/docker.sock -v  
/var/lib/docker:/fenced/mnt/host/var/lib/docker/:rw -v  
/:/fenced/mnt/host/:ro -v /home/user/db:/tmp:rw  
deepfenceio/deepfence_depcheck -u true
```

In the following screenshot we can see the output of the previous command:

Figure 6.12: Initializing vulnerability database

To analyze a specific image, for example, we can download the deepfenceio/fis-java-openshift image from Docker hub repository <https://hub.docker.com/r/deepfenceio/fis-java-openshift> .

```
$ docker pull deepfenceio/fis-javaOpenshift
```

In the following screenshot we can see the output of the previous command:

Figure 6.13: Downloading image from Docker hub

To analyze a specific image we can pass the -t all parameter:

```
$ docker run -ti -v /var/run/docker.sock:/var/run/docker.sock -v  
/var/lib/docker:/fenced/mnt/host/var/lib/docker/:rw -v  
/:/fenced/mnt/host/:ro -v /home/user/db:/tmp:rw  
deepfenceio/deepfence_depcheck -t all -ideepfenceio/fis-javaOpenshift -j true
```

In the following screenshot we can see the output of the previous command:

Figure 6.14: Scanning Docker image for detecting vulnerabilities

At this point we have reviewed OWASP dependency check script for checking vulnerabilities in Docker images.

MicroScanner

MicroScanner is another tool to scan container images for vulnerabilities in packages. It uses the same vulnerability database as the Aqua security commercial scanner.

<https://github.com/aquasecurity/microscanner>

To use MicroScanner, you must first sign up for a token, which can be obtained from the service <https://microscanner.aquasec.com/signup> or by running the command:

```
$ docker run --rm -it aquasec/microscanner --register
```

In the following screenshot you can see the MicroScanner page for registering for a token:



Aqua Security's MicroScanner lets you check your container images for vulnerabilities. If your image has any known high-severity issue, MicroScanner can fail the image build, making it easy to include as a step in your CI/CD pipeline.

Registering for a token

Figure 6.15: MicroScanner page for registering token

MicroScanner is designed to run as part of building an image of a container. To start using it, simply add the following code snippet in the Dockerfile of the image you want to analyze and add the key that we have obtained when registering.

```
ADD https://get.aquasec.com/microscanner .
```

```
RUN chmod +x microscanner
```

```
RUN ./microscanner
```

For example, the following commands correspond to a Dockerfile file that performs the construction of an image based on python, on which it executes MicroScanner.

```
FROM python:3.7
```

```
RUN apt-get update && apt-get -y install ca-certificates
```

```
ADD https://get.aquasec.com/microscanner/
```

```
RUN chmod +x /microscanner
```

```
ARG token
```

```
RUN /microscanner ${token}
```

```
RUN echo "No vulnerabilities!"
```

These commands download the MicroScanner binary, give it permission to run and execute it on the contents of the file system of the base image.

```
$ docker build --build-arg=token= --no-cache.
```

In the following screenshot, you can see the output of the previous command:

Figure 6.16: MicroScanner execution build command

At this point, we have reviewed the MicroScanner script for checking vulnerabilities in docker images. If MicroScanner finds a vulnerability, it reports the details in JSON format.

Clair scanner and quay.io repository

CoreOS Clair is an open-source project for static vulnerability analysis in container-based applications. As layers can be shared among many containers, introspection is vital to create a package inventory and compare it with known CVEs. Since Clair's image analysis is static, it is never necessary for the containers to actually run, so the analysis time is greatly reduced and does not require that potentially vulnerable containers be run.

At the analysis level, it performs the following checks:

Scan an image against Clair's server

Compare vulnerabilities against a white list

It tells you if there are vulnerabilities that are not whitelisted

By extracting static information from the image file system and maintaining a list of differences between the different layers of which the image is composed, the analysis time is greatly reduced and does not require that potentially vulnerable containers be executed.

Clair provides a JSON API that extracts all layers of the image and can be run locally to inspect container images, for example, as part of a process of continuous integration and delivery.

In the following screenshot you can see the Linux distributions supported by Clair scanner:

Data Source	Data Collected	Format	License
Debian Security Bug Tracker	Debian 6, 7, 8, unstable namespaces	dpkg	Debian
Ubuntu CVE Tracker	Ubuntu 12.04, 12.10, 13.04, 14.04, 14.10, 15.04, 15.10, 16.04 namespaces	dpkg	GPLv2
Red Hat Security Data	CentOS 5, 6, 7 namespaces	rpm	CVRF
Oracle Linux Security Data	Oracle Linux 5, 6, 7 namespaces	rpm	CVRF
Alpine SecDB	Alpine 3.3, Alpine 3.4, Alpine 3.5 namespaces	apk	MIT
NIST NVD	Generic Vulnerability Metadata	N/A	Public Domain

Figure 6.17: Linux distributions supported by Clair scanner

To install Clair, we can do it through the Docker Compose tool and the repository <https://github.com/hxquangnhatclair-analyze-local-images>.

In the following screenshot you can see the GitHub repository for Clair scanner:

clair_config	clair	2 years ago
Dockerfile	clair	2 years ago
README.MD	clair	2 years ago
docker-compose.yml	clair	2 years ago
README.MD		

Clair CoreOS with local-image-analyzer ready



clair

A Dockerfile and docker-compose for running Vulnerability Static Analysis for Containers using [Clair](#).

The Dockerfile also contains the local-image-analyzer for analyzing local docker images.

Figure 6.18: GitHub repository for Clair scanner

The installation instructions are:

```
$ git clone https://github.com/hxquangnhatclair-analyze-local-images.git
```

```
$ cd clair-analyze-local-images/
```

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

This repository contains the following file that we can execute with the docker-compose up command. This is the content of the docker-compose where we see the services of Postgres and Clair:

```
docker-compose.yml
```

In the following screenshot, you can see the content of this file:

```

1 version: '2'
2 services:
3   postgres:
4     container_name: clair_postgres
5     image: postgres:latest
6     environment:
7       POSTGRES_PASSWORD: password
8
9   clair:
10    container_name: clair_clair
11    image: hxquangnhat/clair:latest
12    depends_on:
13      - postgres
14    ports:
15      - "6060-6061:6060-6061"
16    links:
17      - postgres
18    volumes:
19      - /tmp:/tmp
20      - ./clair_config:/config
21      - /var/run/docker.sock:/var/run/docker.sock
22    command: [-config, /config/config.yaml]

```

Figure 6.19: Docker-compose file for Clair scanner

Here we see the 2 containers in execution, and to analyze an image, we execute the container with name Clair. When executing previous commands, we have 2 containers running, one corresponding to the Postgres database listening on port 5432 and another corresponding to the image analyzer listening on port 6061 .

In the following screenshot you can see 2 containers for Clair and Postgres services in running state:

Figure 6.20: Containers for Clair and Postgres services in running state

For analyzing vulnerabilities in an image, we can use the following command:

```
$ docker exec clair_clairanalyser
```

If the image has vulnerabilities, it shows the corresponding CVE:

CVE-2017-11755 (Negligible)

The WritePICONImage function in coders/xpm.c in ImageMagick 7.0.6-4 allows remote attackers to cause a denial of service (memory leak) via a crafted file that is mishandled in an AcquireSemaphoreInfo call.

Package: imagemagick @ 8:6.8.9.9-5+deb8u14
Link: <https://security-tracker.debian.org/tracker/CVE-2017-11755>
Layer: 03436a56679d1d5ded06fee217047b4cc7f753913bc5073fe44461cf87e954e4

CVE-2017-11166 (Negligible)

The ReadXWDImage function in coders\xwd.c in ImageMagick 7.0.5-6 has a memory leak vulnerability that can cause memory exhaustion via a crafted length (number of color-map entries) field in the header of an XWD file.

Package: imagemagick @ 8:6.8.9.9-5+deb8u14
Link: <https://security-tracker.debian.org/tracker/CVE-2017-11166>
Layer: 03436a56679d1d5ded06fee217047b4cc7f753913bc5073fe44461cf87e954e4

CVE-2017-13062 (Negligible)

In ImageMagick 7.0.6-6, a memory leak vulnerability was found in the function formatIPTC in coders/meta.c, which allows attackers to cause a denial of service (WriteMETAIImage memory consumption) via a crafted file.

Figure 6.21: Vulnerabilities found in a Docker image

For more information about the installation and execution of the tool, we can query the official documentation:

<https://github.com/coreosclair/blob/master/Documentation/running-clair.md>.

Another alternative for scanning vulnerabilities in images and Docker containers, we can use the Clair tool, from CoreOS, to do this, we will use the following commands for deployment:

The first step is to pull a container from the Postgres version 9.6 database:

```
$ sudo docker run -d -e POSTGRES_PASSWORD="" -p 5432:5432 --restart always postgres:9.6
```

In the following screenshot we can see the output of the previous command:

```
dockerserverclair@DockerServerClair:~$ sudo docker run -d -e POSTGRES_PASSWORD="" -p 5432:5432 --restart always postgres:9.6
809024d7af2a7924e9c71244cf06aaaf2854d5274495bb7b450f805bb9945c78d
dockerserverclair@DockerServerClair:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
809024d7af2a        postgres:9.6      "docker-entrypoint.s..."   10 seconds ago    Up 9 seconds       0.0.0.0:5432->5432/tcp
cranky_shtern
dockerserverclair@DockerServerClair:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
809024d7af2a        postgres:9.6      "docker-entrypoint.s..."   12 seconds ago    Up 11 seconds      0.0.0.0:5432->5432/tcp
cranky_shtern
dockerserverclair@DockerServerClair:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
```

Figure 6.22: Step 1 command

In the second step, we create a Clair configuration file with the commands:

```
$ sudomkdir $HOME/clair_config
```

```
$ sudo curl -L http://raw.githubusercontent.com/coreosclair/master/config.yaml.sample -o $PWD/clair_config/config.yaml
```

In the following screenshot we can see the output of the previous commands:

```

dockerserverclair@DockerServerClair:~$ sudo mkdir $HOME/clair_config
dockerserverclair@DockerServerClair:~$ sudo curl -L https://raw.githubusercontent.com/coreos/clair/master/config.yaml.sample -o $PWD/clair_config/config.yaml
  % Total    % Received % Xferd  Average Speed   Time     Time      Time  Current
          Dload  Upload   Total Spent   Left  Speed
100 2941  100 2941    0     0 17300      0 --:--:-- --:--:-- 17402
dockerserverclair@DockerServerClair:~$ ls
clair_config
dockerserverclair@DockerServerClair:~$ █

```

Figure 6.23: Step 2 commands

In the URL <https://raw.githubusercontent.com/coreos/clair/master/config.yaml.sample> , we can find the YAML file configuration for deploying Clair services with Docker.

In the third step, we download Clair container and check Clair state:

```
$ sudo docker run --net=host -d -p 6060-6061:6060-6061 --restart always -v $PWD/clair_config:/config quay.io/coreos/clair:v2.0.7 --config=/config/config.yaml
```

```
$ curl -X GET -I http://localhost:6061/health
```

In the following screenshot we can see the output of the previous commands:

```

dockerserverclair@DockerServerClair:~$ sudo docker run --net=host -d -p 6060-6061:6060-6061 --restart always -v $PWD/clair_config:/config quay.io/coreos/clair:v2.0.7 --config=/config/config.yaml
WARNING: Published ports are discarded when using host network mode
2621447b1ladcb3c87e5bc305886652fb1e81825783de2146d28bd3ba8b96cac
dockerserverclair@DockerServerClair:~$ curl -X GET -I http://localhost:6061/health
HTTP/1.1 200 OK
Server: clair
Date: Sun, 25 Nov 2018 16:23:48 GMT
Content-Length: 0

dockerserverclair@DockerServerClair:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
TS                  NAMES
2621447b1lad        quay.io/coreos/clair:v2.0.7   "/clair -config=/con..."   50 seconds ago    Up 49 seconds
809024d7af2a        postgres:9.6           "docker-entrypoint.s..."   2 minutes ago     Up 2 minutes
.0.0:5432->5432/tcp  cranky_shtern

dockerserverclair@DockerServerClair:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
TS                  NAMES

```

Figure 6.24: Step 3 commands

In the fourth step, we install and configure Clair scanner:

```
$ mkdir clairscanner
```

```
$ sudowget
```

```
$ http://github.com/arminc/clair-scanner/releases/download/v8/clair-scanner/linuxamd64
```

```
$ sudo mv clair-scanner/linuxamd64 clair-scanner
```

```
$ sudo chmod +x clair-scanner
```

In the following screenshot we can see the output of the previous commands:

```

dockerserverclair@DockerServerClair:~$ mkdir clairscanner
dockerserverclair@DockerServerClair:~$ cd clairscanner
dockerserverclair@DockerServerClair:~/clairscanner$ sudo wget https://github.com/arminc/clair-scanner/releases/download/v8/clair-scanner_linux_amd64
--2018-11-25 16:25:01--  https://github.com/arminc/clair-scanner/releases/download/v8/clair-scanner_linux_amd64
Resolving github.com (github.com)... 140.82.118.3, 140.82.118.4
Connecting to github.com (github.com)|140.82.118.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://github-production-release-asset-2e65be.s3.amazonaws.com/86972405/4061695e-f44f-11e7-97fe-da8073f4908
c?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20181125%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20181125T162502Z&X-Amz-Expires=300&X-Amz-Signature=20f44d5d109863ab5f22e56d92c5579342bc17b04f384086af2b3064a5
4d7bec&X-Amz-SignedHeaders=host&actor_id=0&response-content-disposition=attachment%3B%20filename%3Dclair-scanner_linux_amd64&response-content-type=application%2Foctet-stream [following]
--2018-11-25 16:25:02--  https://github-production-release-asset-2e65be.s3.amazonaws.com/86972405/4061695e-f44f-11e7-97fe-da8073f4908c?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20181125%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20181125T162502Z&X-Amz-Expires=300&X-Amz-Signature=20f44d5d109863ab5f22e56d92c5579342bc17b04f384086af2b3064a5
4d7bec&X-Amz-SignedHeaders=host&actor_id=0&response-content-disposition=attachment%3B%20filename%3Dclair-scanner_linux_amd64&response-content-type=application%2Foctet-stream
Resolving github-production-release-asset-2e65be.s3.amazonaws.com (github-production-release-asset-2e65be.s3.amazonaws.com)... 52.216.109.155
Connecting to github-production-release-asset-2e65be.s3.amazonaws.com (github-production-release-asset-2e65be.s3.amazonaws.com)|52.216.109.155|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9862522 (9.4M) [application/octet-stream]
Saving to: 'clair-scanner_linux_amd64'

clair-scanner_linux_amd64      100%[=====]  9.41M  11.5MB/s   in 0.8s

2018-11-25 16:25:03 (11.5 MB/s) - 'clair-scanner_linux_amd64' saved [9862522/9862522]

dockerserverclair@DockerServerClair:~/clairscanner$ ls
clair-scanner linux amd64

```

Figure 6.25: Step 4 commands

In the following screenshot we can see how we give Clair-scanner execution permissions with chmod command:

```

dockerserverclair@DockerServerClair:~/clairscanner$ sudo mv clair-scanner_linux_amd64 clair-scanner
dockerserverclair@DockerServerClair:~/clairscanner$ ls
clair-scanner
dockerserverclair@DockerServerClair:~/clairscanner$ sudo chmod +x clair-scanner
dockerserverclair@DockerServerClair:~/clairscanner$ ls
clair-scanner
dockerserverclair@DockerServerClair:~/clairscanner$ 

```

Figure 6.26: Clair-scanner execution permissions

In the fifth step, we pull an image that we know has vulnerabilities.

For example, the image available in Docker Hub <https://hub.docker.com/r/vulnerables/cve-2016-10033> has a remote code execution vulnerability.

```
$ sudo docker pull vulnerables/cve-2016-10033
```

In the following screenshot we can see the output of the previous command:

```

dockerserverclair@DockerServerClair:~/clairscanner$ sudo docker pull vulnerables/cve-2016-10033
Using default tag: latest
latest: Pulling from vulnerables/cve-2016-10033
85b1f47fba49: Pull complete
fd1b7848a24f: Pull complete
958cf444071: Pull complete
1bce47ee484d: Pull complete
b0d2bcaa617: Pull complete
401c3f47975f: Pull complete
52c46abb054a: Pull complete
Digest: sha256:271282ecaa7728ceb618515e389ce92bc884ee24f03ba65bf134d85f72a86279
Status: Downloaded newer image for vulnerables/cve-2016-10033:latest
dockerserverclair@DockerServerClair:~/clairscanner$ sudo docker images
REPOSITORY          TAG           IMAGE ID        CREATED       SIZE
quay.io/coreos/clair    v2.0.7      7968a46189ae   5 days ago   353MB
postgres            9.6         036ff0ef319f   9 days ago   229MB
vulnerables/cve-2016-10033  latest     6f50d85ccbe1  13 months ago  242MB
dockerserverclair@DockerServerClair:~/clairscanner$ 

```

Figure 6.27: Step 5 command

In the sixth step, we launch the analysis of the unloaded container using Clair scanner:

```
$ sudo ./clair-scanner -c http://localhost:6060 vulnerable/cve-2016-10033
```

In the following screenshot we can see the output of the previous command:

```
2018/11/25 16:28:21 [INFO] □ Analyzing dde8e59bffb1df7fc5f6d4a60dcaa2a7d936eab1fe455a9f8dd96b64b706b37
2018/11/25 16:28:24 [INFO] □ Analyzing ffc629ef32caddb37507405b90fe425e1bdaf77d087832ea68e3fee5d079516e
2018/11/25 16:28:24 [INFO] □ Analyzing e90cc14beb2b09c943dd6b4a55fb3c2c14fb06ae38c5308f24f87f2faf45a4b
2018/11/25 16:28:24 [INFO] □ Analyzing f7ee4dae102e919cd5e29ee3402d36c8736888ec16a16398957df2674019c3ec
2018/11/25 16:28:24 [INFO] □ Analyzing e1c8276f5ff4ff62f553edf8d749a594c8fb92f13485f45ddf37d7c5348e144
2018/11/25 16:28:24 [INFO] □ Analyzing 512148149fabf6ca5calbcfc6ef3c3c8d2b3076746d5b74719011a8208eb796b
2018/11/25 16:28:24 [INFO] □ Analyzing 70d3771c19ba4c1b4ee59092a35eeecc79187793b9680e4b3e4c6e7579e5cf6
2018/11/25 16:28:24 [WARN] □ Image [vulnerable/cve-2016-10033] contains 233 total vulnerabilities
2018/11/25 16:28:24 [ERRO] □ Image [vulnerable/cve-2016-10033] contains 233 unapproved vulnerabilities
+-----+-----+-----+
| STATUS | CVE SEVERITY | PACKAGE NAME | PACKAGE VERSION | CVE DESCRIPTION
+-----+-----+-----+
| Unapproved | High CVE-2016-2779 | util-linux | 2.25.2-6 | runuser in util-linux allows local users to escape to the parent session via a crafted TIOCSTI ioctl call, which pushes characters to the terminal's input buffer.
```

Figure 6.28: Step 6 command

In the following screenshot, we can see vulnerabilities detected in the Docker image vulnerable/cve-2016-10033 :

software vulnerable to timing attacks				local users to exploit s
g attack on 'port contention'.				via a side-channel timin
.debian.org/tracker/CVE-2018-5407				https://security-tracker
-----+	-----+	-----+	-----+	-----+
Unapproved Unknown CVE-2018-14647	python2.7	2.7.9-2+deb8u1	Python's elementtree C a	
ccelerator failed to initialise				Expat's hash salt during
initialization. This could make				it easy to conduct denia
l of service attacks against				Expat by constructing an
XML document that would cause				pathological hash collis
ions in Expat's internal data				structures, consuming la
rge amounts CPU and RAM. Python				3.8, 3.7, 3.6, 3.5, 3.4,
2.7 are believed to be vulnerable.				https://security-tracker
.debian.org/tracker/CVE-2018-14647				
-----+	-----+	-----+	-----+	-----+
Unapproved Unknown CVE-2018-19518	php5	5.6.30+dfsg-0+deb8u1	https://security-track	
er.debian.org/tracker/CVE-2018-19518				
-----+	-----+	-----+	-----+	-----+

Figure 6.29: Vulnerabilities detected in the Docker image vulnerable/cve-2016-10033

We could also analyze Docker images with the clairctl service available in the GitHub repository.

<https://github.com/jgsquare/clairctl>

In the same way that we have seen before, we have a Docker compose file docker-compose.yml that runs 3 services (Postgres, Clair, and clairctl):

version: '2.1'

services:

postgres:

```
image: postgres:9.6
restart: unless-stopped
volumes:
  - ./docker-compose-data/postgres-data:/var/lib/postgresql/data:rw
```

```
environment:
  - POSTGRES_PASSWORD=ChangeMe
  - POSTGRES_USER=clair
  - POSTGRES_DB=clair
```

clair:

```
image: quay.io/coreos/clair:v2.0.0
```

```
restart: unless-stopped
```

volumes:

```
  - ./docker-compose-data/clair-config:/config:ro
  - ./docker-compose-data/clair-tmp:/tmp:rw
```

depends_on:

postgres:

```
condition: service_started
```

```
command: [--log-level=debug, --config, /config/config.yml]
```

clairctl:

```
image: jgsquare/clairctl:latest
```

```
restart: unless-stopped
```

environment:

```
  - DOCKERAPIVERSION=1.24
```

volumes:

```
  - ./docker-compose-data/clairctl-reports:/reports:rw
  - /var/run/docker.sock:/var/run/docker.sock:ro
```

depends_on:

clair:

```
condition: service_started
```

When executing the docker-compose command over the previous file, we see how we have 3 containers running, one for each of the services we have in the docker-compose.yml file.

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

In the following screenshot we can see the output of the previous command:

Figure 6.30: Execution of the docker-compose command

With the docker ps command, we can see the services related to Clair and Postgres that have been deployed.

In this way, we have the clairctl command available to execute it as a Docker image.

Figure 6.31: Execution of the docker-compose exec clair

To perform an analysis of an image, we can execute the command:

```
$ docker-compose exec clairctl clairctl analyze --local
```

At this point, we have reviewed the execution of Clair using Docker compose and how we can detect vulnerabilities in a specific Docker image.

Github repositories and Clair links

Among the main repositories of GitHub and articles with more information about the tool we can highlight:

<https://github.com/coreos/clair>

<https://github.com/arminc/clair-scanner>

<https://github.com/arminc/clair-local-scan>

<https://bitbucket.org/osallou/clair>

<https://coreos.com/blog/vulnerability-analysis-for-containers.html>

<https://thenewstack.io/coreos-introduces-container-scanning-for-vulnerabilities>

In this section, we have reviewed the installation and execution of Clair for scanning Docker images. In the next section, we will review the Quay.io image repository for static image analysis.

Quay.io image repository

CoreOs, in addition to using Clair to analyze Docker images uploaded by customers, also uses the Quay.io image repository <https://docs.quay.io> , a container registry with similar features to the Docker hub repository.

Quay registry provides static image analysis with the objective of finding obsolete and vulnerable libraries in binaries. Among the main features we can highlight:

Scanning the image before uploading it to production

Vulnerability mapping with CVE updates

Links to the vulnerabilities of the CVE database

Easy to integrate into your CI/CD workflow

Web interface and analysis engine API

At low level uses the CoreOS Clair open source CVE engine

We continue with the registration and login into Quay.io service.

Register in Quay.io

To log in to Quay.io , you must execute the docker login quay.io command from the command line. The following steps can be found in <https://docs.quay.io/solution/getting-started.html>

\$ docker login quay.io

Username: myusername

Password: mypassword

Within the configuration of the Quay.io account, it is possible to create an encrypted password for added security. The next step would be to create a new container on which to execute the image that we are going to analyze.

When creating the container, we see that it returns an identifier:

Create a new container

First we'll create a container with a single new file based off of the `ubuntu` base image:

```
$ docker run ubuntu echo "fun" > newfile
```

The container will immediately terminate (because its one command is `echo`), so we'll use `docker ps -l` to list it:

CONTAINER ID	IMAGE	COMMAND	CREATED
07f2065197ef	ubuntu:12.04	echo fun	31 seconds ago

Make note of the `container id`; we'll need it for the commit command.

Figure 6.32: Creating a new container, it will return a container identifier

As the last step, we must create the name of the repository where we will subsequently upload the image:

```
$ docker commit quay.io/username/reponame
```

When creating the repository name, we see that it returns an identifier:

Tag the container to an image

We next need to tag the container to a known image name

Note that the `username` must be your Quay.io username and `reponame` is the new name of your repository.

```
$ docker commit 07f2065197ef quay.io/username/reponame  
e7050e05a288f9f3498ccd2847fee966d701867bc671b02abf03a6629dc921bb
```

Figure 6.33: Tag the container to an image

To send an image to a repository in Quay.io, we need to execute the `docker push` command:

```
$ sudo docker push quay.io/namespace/repository:tag
```

The push refers to a repository [quay.io/username/reponame] (len: 1)

Sending image list

Pushing repository quay.io/username/reponame (1 tags)

```
8dbd9e392a96: Pushing [=====> ] 21.27 MB/134.1 MB 40s
```

To extract an image from the Quay.io repository, run the following command:

```
$ sudo docker pull quay.io/namespace/repository
```

Another way to create a repository is through the Quay.io user web interface. Click on the + icon in the upper right corner and select New repository . In the following screenshot we can see quay.io page for creating a new repository:

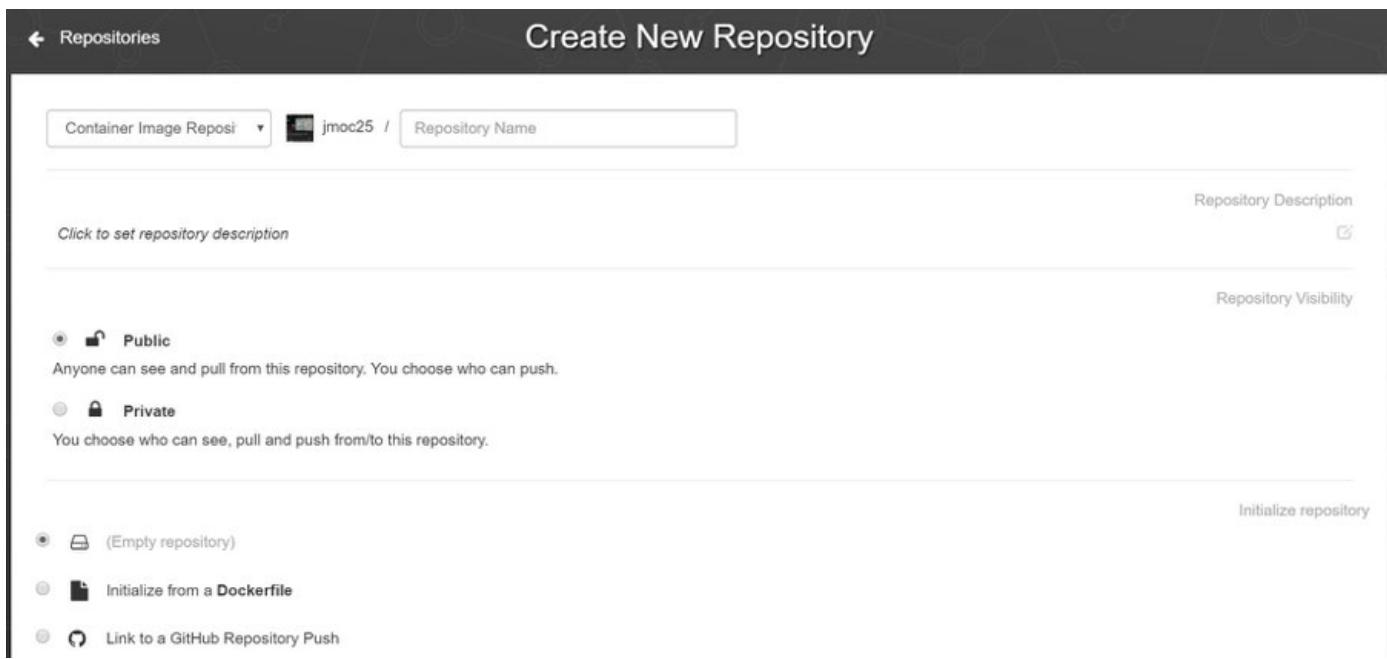


Figure 6.34: Quay.io page for creating a new repository

The tags of a repository can be viewed and modified in the label panel of the repository page.

In the following screenshot we can see Quay.io page for viewing repository tags:

Figure 6.35: Quay.io page for viewing repository tags

From this interface, it is possible to assign several labels to the same image. A new tag can be added to a tagged image by clicking on the icon next to the tag and selecting Add new tag. Quay.io will confirm the action of adding a new label to the image.

In the following screenshot we can see actions related to tags:

Repository Tags

Compact Expanded

TAG	LAST MODIFIED	SECURITY SCAN	SIZE	EXPIRES	MANIFEST
latest	5 minutes ago	Unsupported	20.4 MB	Never	SHA256 0fe333c46ff4

- [+ Add New Tag](#)
- [Edit Labels](#)
- [Delete Tag](#)
- [Change Expiration](#)

Figure 6.36: Actions related to tags

In the Manifest layers section, we can see the layers that make up the image:

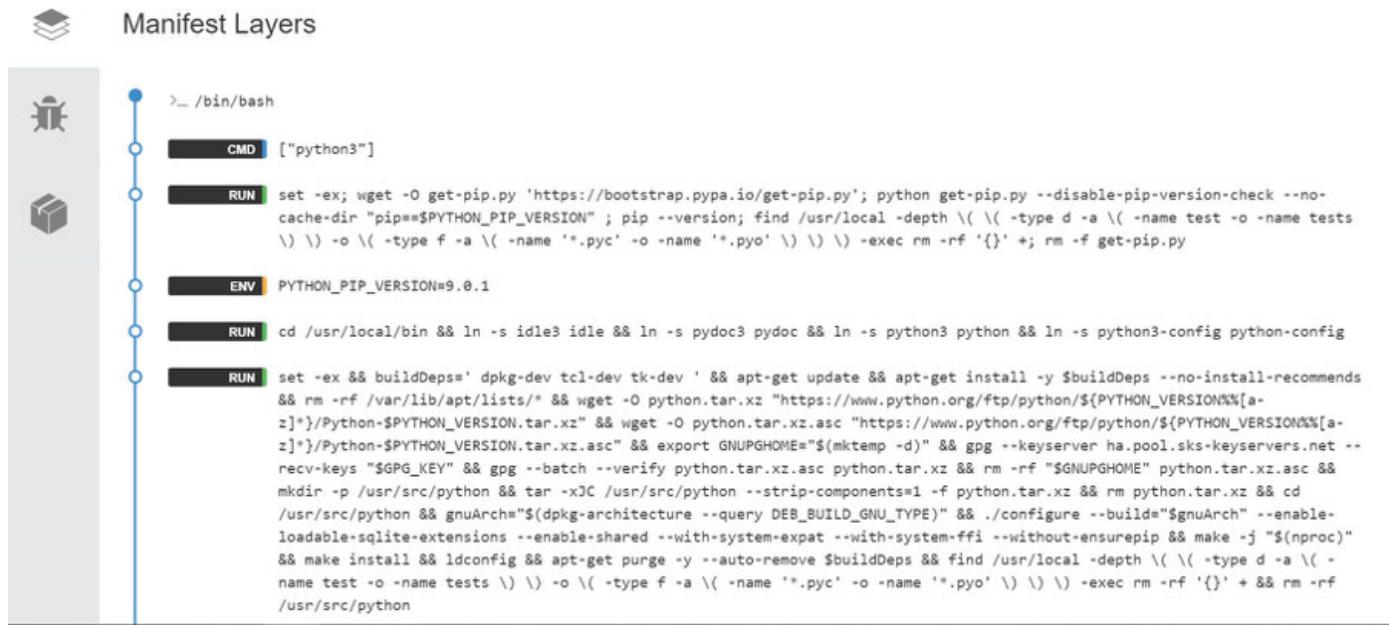


Figure 6.37: Manifest layers

We can see information related to the image scan, including the vulnerabilities that have been detected in each of the layers. For each of the vulnerabilities, it shows the CVE number, the level of criticality, affected package, a version that contains the vulnerability, and the version that has the fix and could solve the security flaw.

In the following screenshot we can see vulnerabilities detected by Quay security scanner:

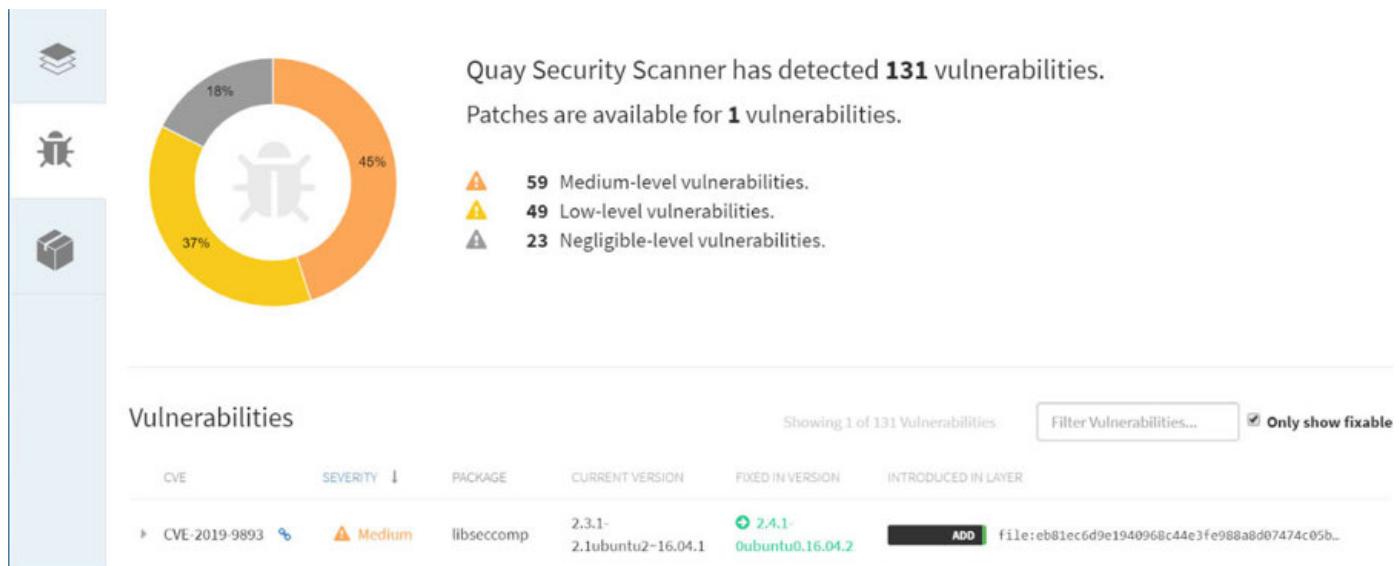


Figure 6.38: Vulnerabilities detected by Quay security scanner

In the following screenshot, we can see image vulnerabilities with the corresponding CVE:

The figure shows the "Image Vulnerabilities" section. It lists 31 out of 569 vulnerabilities, each with a CVE link, severity (High), package, current version, fixed version, and a "RUN" button with a command. The table columns are:

CVE	SEVERITY	PACKAGE	CURRENT VERSION	FIXED IN VERSION	INTRODUCED IN IMAGE
CVE-2017-17805	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2017-17558	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2018-2562	⚠️ High	mysql-5.5	5.5.58-0+deb8u1	5.5.59-0+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2017-16538	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2017-8824	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2017-17806	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2017-16939	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...
CVE-2017-15868	⚠️ High	linux	3.16.51-2	3.16.51-3+deb8u1	RUN set -ex; apt-get update; apt...

Figure 6.39: Image vulnerabilities with the corresponding CVE

A high, medium, or low criticality levels are assigned to vulnerabilities in the scan report. This criticality level depends on the CVSS score:

High: The vulnerability has a basic CVSS score that varies from 8.0 to 10.0.

Medium: The vulnerability has a CVSS base score ranging from 4.0-7.9.

Low: The vulnerability has a basic CVSS score ranging from 0.0 to 3.9.

If we go into details, we see that for each vulnerability, a series of metrics is defined that will give the final score and the level of criticality:

Vulnerabilities

Filter Vulnerabilities..

CVE	SEVERITY ↓	PACKAGE	CURRENT VERSION	FIXED IN VERSION	INTRODUCED IN LAYER
▶ CVE-2016-4448	10 / 10	libxml2	2.9.1+dfsg1-5+deb8u5	(None)	set -ex; apt-get update
▶ CVE-2017-17458	10 / 10	mercurial	3.1.2-2+deb8u4	3.1.2-2+deb8u6	apt-get update && apt
▼ CVE-2017-18017	10 / 10	linux	3.16.51-2	3.16.56-1	set -ex; apt-get update

VECTORS

Access Vector	Access Complexity	Authentication	Confidentiality Impact	Integrity Impact
Network	Low	None	Complete	Complete
Adjacent Network	Medium	Single	Partial	Partial
Local	High	Multiple	None	None

DESCRIPTION

The `tcpmss_mangle_packet` function in `net/netfilter/xt_TCPMSS.c` in the Linux kernel before 4.11, and 4.9.x before 4.9.36, allows remote attackers to cause a denial of service (use-after-free and memory corruption) or possibly have unspecified other impact by leveraging the presence of `xt_TCPMSS` in an iptables action.

Figure 6.40: Vulnerabilities detected by Quay security scanner

In the previous image, we see that each vulnerability defines a series of metrics that will give the final score and the level of criticality.

Access vector: This metric reflects how the vulnerability is exploited. The more remote an attacker is to attack a host, the higher the vulnerability score.

In the previous example, we see that the value of this metric is the network. This metric measures the impact on the integrity of a successfully exploited vulnerability. Integrity refers to the reliability and veracity of the information. The increase in the impact of integrity increases the vulnerability score.

Access complexity: This metric measures the complexity of the attack required to exploit the vulnerability once an attacker has accessed the target system.

Authentication: This metric measures the strength or complexity of the authentication process; for example, if an attacker is required to provide credentials before he can run an exploit. The fewer authentication instances required, the higher the vulnerability score.

Confidentiality impact: Confidentiality refers to limiting access to information and disclosure only to authorized users, as well as preventing access or disclosure to unauthorized persons. Increasing the impact of confidentiality increases the vulnerability score.

Integrity impact: This metric measures the impact on the integrity of a successfully exploited vulnerability. The increase in the impact of integrity increases the vulnerability score.

You can find more information about these metrics in the URL: <https://www.first.org/cvss/specification-document>.

Analyzing Docker images with Anchore engine and Anchore CLI

Anchore is an open-source solution, available in on-premise version or in a service version managed by third parties, where its main objective is the discovery of Docker images in both public and private repositories, with the objective of performing a static analysis of known vulnerabilities.

This solution allows the list of packages installed in the operating system of the docker image, as well as the complete list of all Node.js and Ruby files and dependencies included in, said Docker image. The static analysis of known vulnerabilities will be performed on the set of packages installed at the operating system level (base image).

In the following image, we can see Anchore engine architecture:

Figure 6.41: Anchore engine architecture

The architecture of the anchor engine consists of six components that can be implemented in a single container or in a Kubernetes cluster:

API service: This is a central messaging interface that can be accessed via a script using a REST API or using the command line directly.

Image analyzer service: This is the service executed by the worker nodes that perform the process of scanning Docker images.

Catalog service: Internal database that contains the catalog of vulnerabilities.

Waiting service: This organizes, persists, and schedules internal engine tasks.

Policy Engine Service: This provides vulnerability analysis in accordance with policies and rules.

Kubernetes Webhook Service: Services that allows validating Docker images before they are generated.

Basically, the Anchore engine is provided as a Docker image that can be used with other orchestration platforms such as Kubernetes, Docker Swarm, or Rancher.

Anchore Engine allows developers to perform a detailed analysis of images, execute queries, generate reports, and define policies that can be used in the CI/CD cycle. Anchore offers the possibility of connecting to a Jenkins pipeline, so when a developer uploads code in a repository like GitHub, it activates Jenkins to start a compilation that creates a container image, etc.

The open-source version is highly customizable and reusable for different tasks, from CD/CI tasks to forensic analysis or inspection and debugging tasks. Among other things, it allows:

Extract packages and components from Docker images.

Scan images for known vulnerabilities.

We continue with the installation of the anchore engine with Docker compose.

Starting Anchore engine

There are several ways to start the anchore-engine; one of the most direct and simple is through the compose.yaml docker where all the services that are needed are declared. To do this, we can use docker-compose.yaml file that we find within the scripts/docker-compose directory in the GitHub project.

<https://github.com/anchore/anchore-engine/blob/master/docker-compose.yaml>

These are the command you can use for downloading the last version of anchore-engine :

```
$ git clone https://github.com/anchore/anchore-engine
```

```
$ cd anchore-engine
```

In the following screenshot we can see the output of the previous commands:

```

$ git clone https://github.com/anchore/anchore-engine
Cloning into 'anchore-engine'...
remote: Enumerating objects: 121, done.
remote: Counting objects: 100% (121/121), done.
remote: Compressing objects: 100% (82/82), done.
remote: Total 12804 (delta 51), reused 80 (delta 35), pack-reused 12683
Receiving objects: 100% (12804/12804), 20.92 MiB | 11.25 MiB/s, done.
Resolving deltas: 100% (8515/8515), done.
Checking out files: 100% (985/985), done.
[node1] (local) root@192.168.0.53 ~
$ cd anchore-engine
[node1] (local) root@192.168.0.53 ~/anchore-engine
$ ls
bash: LS: command not found
[node1] (local) root@192.168.0.53 ~/anchore-engine
$ ls
CHANGELOG.md          anchore_manager      requirements-test.txt
CONTRIBUTING.rst       conf                requirements.txt
Dockerfile             docker-compose-dev.yaml scripts
LICENSE                docker-compose.yaml   setup.py
MANIFEST.in            docker-entrypoint.sh test

```

Figure 6.42: Downloading Anchore engine source code

To start Anchore Engine, we can execute the docker compose -d command on the same path where we have downloaded the docker-compose.yaml file.

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

In the following screenshot we can see the output of the previous command:

```

$ docker-compose up -d
Starting anchore-engine_anchore-db_1 ... done
Starting anchore-engine_engine-catalog_1 ... done
Starting anchore-engine_engine-api_1           ... done
Starting anchore-engine_engine-policy-engine_1 ... done
Starting anchore-engine_engine-simpleq_1        ... done
Starting anchore-engine_engine-analyzer_1       ... done

```

Figure 6.43: Starting Anchore engine containers

Here, we can see the anchore-engine containers running each in its corresponding port. We see that the anchore-engine service is downloading the image from the docker.io/anchore/anchore-engine repository, and also this service depends on the anchore-db service, which in turn downloads the Postgres image, establishing the volumes of persistence and the necessary environment variables.

In the following screenshot we can see Anchore engine containers in execution:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
	PORTS	NAMES		
80c972373c0a te (healthy)	anchore/anchore-engine:v0.4.0 8228/tcp	"/docker-entrypoint..." anchore-engine_engine-policy-engine_1	4 minutes ago	Up About a minu
58c2e346f470 te (healthy)	anchore/anchore-engine:v0.4.0 8228/tcp	"/docker-entrypoint..." anchore-engine_engine-analyzer_1	4 minutes ago	Up About a minu
a9b8908bf9c6 te (healthy)	anchore/anchore-engine:v0.4.0 8228/tcp	"/docker-entrypoint..." anchore-engine_engine-simpleq_1	4 minutes ago	Up About a minu
ea4dedb53252 te (healthy)	anchore/anchore-engine:v0.4.0 0.0.0.0:8228->8228/tcp	"/docker-entrypoint..." anchore-engine_engine-api_1	4 minutes ago	Up About a minu
38fd99ba7ce3 te (healthy)	anchore/anchore-engine:v0.4.0 8228/tcp	"/docker-entrypoint..." anchore-engine_engine-catalog_1	4 minutes ago	Up About a minu
97f176ece076 te	postgres:9 5432/tcp	"docker-entrypoint.s..." anchore-engine_anchore-db_1	4 minutes ago	Up About a minu

Figure 6.44: Anchore engine containers in execution

The installation of anchore cli could be done in several ways, and the most direct is through the pip install command or through the source code.

```
$ pip install anchorecli

$ git clone https://github.com/anchore/anchore-cli

$ cd anchore-cli

$ pip install --user --upgrade.

$ python setup.py install
```

To execute the anchore container, we can do it from the docker pull anchore/cli command. At this point, you can execute any of the queries that are included with anchore, create and evaluate images against custom policies that are configured to ensure that your containers meet the requirements defined in the security policies. For more information on the above features, see the anchore documentation.
<https://github.com/anchore/anchore/wiki> .

Anchore cli has the ability to communicate with the anchore engine to analyze the images that we have locally on the host docker. It provides a command-line interface at the top of the REST API of the anchore engine. Using the Anchore CLI, users can manage and inspect images, policies, and so on.

In the following screenshot we can see the options of anchore-cli after installing:

Figure 6.45: Anchore cli options

In the following screenshot we can see the commands supported by anchore-cli:

Figure 6.46: Anchore cli commands

These are some of the commands to perform an analysis of an image.

```
$ anchore-cli --u admin --p foobar image
```

In the following screenshot we can see the output of the previous command:

Figure 6.47: Anchore cli commands

These are the commands that could be most useful for analyzing our images:

Add an image to Anchore to analyze

```
$ anchore-cli image add
```

Display image content

```
$ anchore-cli image content os
```

Analyze image content

```
$ anchore-cli image content files
```

Evaluate based on policy compliance

```
$ anchore-cli evaluate check os
```

The following command will return information about the process of analyzing a Docker image:

```
$ anchore-cli --u admin --p foobar image add docker.io/library/alpine:3.4
```

In the following screenshot we can see the output of the previous command:

Figure 6.48: Anchore cli commands

The following command will return the packages installed inside the Docker image:

```
$ anchore-cli --u admin --p foobar image content docker.io/library/alpine:3.4 os
```

In the following screenshot we can see the output of the previous command:

Figure 6.49: Packages in the alpine docker image

The following command will get metadata from the Docker image:

```
$ anchore-cli --u admin --p foobar image metadata docker.io/library/alpine:3.4 manifest
```

In the following screenshot we can see the output of the previous command:

Figure 6.50: Metadata in the alpine docker image

With the following command, once we have analyzed the image, we can see the detected vulnerabilities and the reference to the corresponding CVE code.

```
$ anchore-cli --u admin --p foobar image vuln docker.io/library/alpine:3.4 all
```

In the following screenshot we can see the output of the previous command:

Figure 6.51: Metadata in the alpine Docker image

The previous command shows information for each package has found a vulnerable version, the level of criticality, and in which version it would be resolved.

Conclusion

In this chapter, we have reviewed some open source tools discovering vulnerabilities in Docker images. In order to minimize the exposure of our images before deploying in a productive environment, it is important to analyze possible vulnerabilities layer by layer that could cause an attacker to take control of the application.

In this chapter, the reader has learned about analyzing the security of docker images layer by layer and discover vulnerabilities through the study of static analysis tools.

In the next chapter, we are going to review topics like Docker container threats and system attacks, which can impact in Docker applications, and what are the main vulnerabilities we can find in Docker images.

Questions

Which is the analysis tool that allows you to scan the images layer by layer, allowing you to analyze several languages such as Java, Python, Node.js, JavaScript, Ruby, and PHP?

Which is the base Linux Docker images Dagda supports?

What provides Clair for analyzing each layer of the container looking for existing vulnerabilities in Debian, Ubuntu, and CentOS databases?

Which Docker registry provides static image analysis with the objective of finding obsolete and vulnerable libraries in binaries?

Which are the main components of the anchore engine architecture?

CHAPTER 7

Auditing and Analyzing Vulnerabilities in Docker Containers

This chapter covers topics like Docker container threats and system attacks, which can impact in Docker applications. These threats and attacks are also applicable to specific Docker container versions of the applications. We will review examples of attacks and exploits that could target running containers. Also, we will review specific CVE in Docker images and how we can get details about specific vulnerabilities with vulners API.

In this chapter, the reader will learn about what are the main Docker container threats, what are the main vulnerabilities we can find in Docker images, and some services and tools for getting information about these vulnerabilities. As a result, developers will have the capacity to obtain details about vulnerabilities in container applications.

Structure

Docker containers threats and attacks

Analyzing vulnerabilities in Docker images

CVE in Docker images

Getting CVE details with vulners API

Objectives

Knowing about Docker containers threats and attacks

Knowing about analyzing vulnerabilities in Docker images

Understanding CVE in Docker images

Knowing about obtaining CVE details with vulners API

Docker containers threats and attacks

Among the possible attacks and threats that the containers may suffer, we can highlight direct attacks on the kernel taking advantage of a vulnerability that has not been patched, Denial of Service (DoS) attacks, where the main problem is that the container may monopolize the access to certain resources such as CPU and memory, resulting in denial of service. Another possible attack is the use of trojanized images. If an attacker gets someone to execute a trojanized image with malicious code, both the host docker and the data exposed by it are at risk. Also, it is critical ensuring that images you are running are up-to-date and do not contain software versions with known vulnerabilities.

In this URL, we can see the main vulnerabilities and container attacks related to Docker organized by category:

https://www.cvedetails.com/product/28125/Docker-Docker.html?vendor_id=13534

In this image, we can see the main Docker vulnerabilities organized by category:

Docker » Docker : Vulnerability Statistics

[Vulnerabilities \(23\)](#) [CVSS Scores Report](#) [Browse all versions](#) [Possible matches for this product](#) [Related Metasploit Modules](#)

Related OVAL Definitions : [Vulnerabilities \(0\)](#) [Patches \(2\)](#) [Inventory Definitions \(0\)](#) [Compliance Definitions \(0\)](#)

[Vulnerability Feeds & Widgets](#)

Vulnerability Trends Over Time

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
2014	6		2							1		1			
2015	3										1	1			
2016	2									1		1			
2017	5	2													
2018	2														
2019	5		1						1			1			
Total	23	2	3	0.0	0.0	0.0	0.0	4.3	0.0	8.7	4.3	17.4	0.0	0.0	
% Of All		8.7	13.0	0.0	0.0	0.0									

Figure 7.1: Docker vulnerabilities organized by category

In the following image, we can see the main Docker attacks organized by year and type:

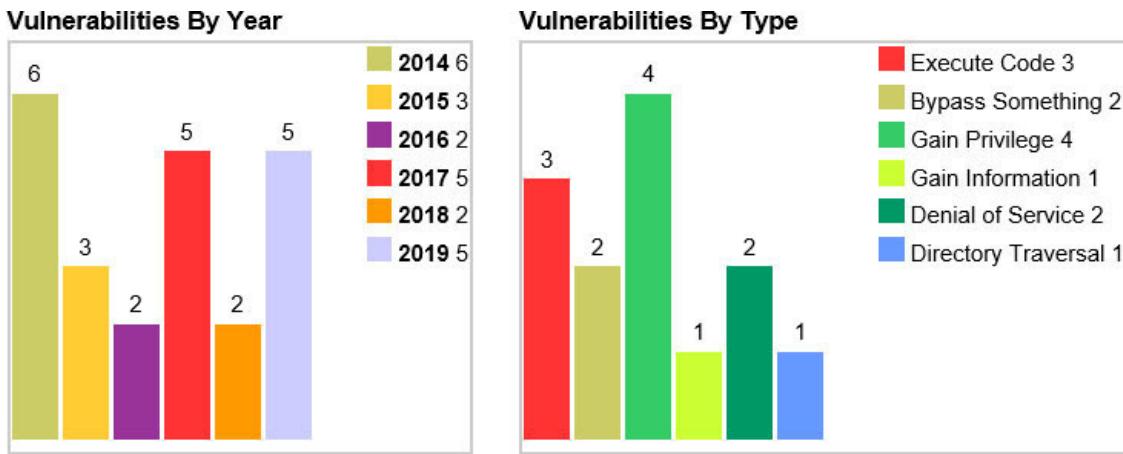


Figure 7.2: Common attacks in Docker containers

Because the containers will always share the host kernel, the container can exploit any vulnerability in the kernel interface to compromise the Docker host, unless it uses seccomp or apparmor to limit calls between the container and the host. Among the threats in the containers, we can highlight:

Application-level DDoS and script attacks between sites in public containers.

Containers that attempt to download additional malware or scan internal systems for vulnerabilities or confidential data.

A container that is forced to use system resources in an attempt to block other containers.

Use of unsecured applications, the aim of which is to deny service on the network and to affect other containers.

The Dirty Cow exploits in the Linux kernel allows root privilege escalation on a host or container.

Ransomware attacks on insecure server containers by MongoDB and Elasticsearch containers.

Buffer overflow vulnerability in Ruby and Python libraries that allow the execution of malicious code.

SQL injection attacks that allow you to take control of a database container in order to steal data.

Vulnerabilities such as buffer overflow based on the glibc stack, which gives control to hackers through man-in-the-middle attacks.

For example, in some python images, it is common to see this type of CVE related to a vulnerability in the glibc library.

Figure 7.3: CVE related to a vulnerability in the glibc library

We could also check if there is an exploit available for this CVE.

Figure 7.4: Exploits available for CVE-2018-1000001

You can find more information about this vulnerability in <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000001> .

Most vulnerabilities associated with CVE are associated with one or more vulnerabilities. For example, CVE-2015-1781, which is a weakness in a buffer overflow that can be abused in DNS servers and leads to denial of service or arbitrary code execution, may fall into three categories: denial of service, execution code and overflow.

One of the most critical CVEs is CVE-2014-9357. This CVE allows remote attackers to execute arbitrary code with root privileges.

Figure 7.5: CVE-2014-9357 details

You can find more information about this vulnerability in <https://www.cvedetails.com/cve/CVE-2014-9357/>.

In this case, the Common Vulnerability Scoring System (CVSS) is being used to measure and compare threats. The common vulnerability scoring system is based on factors such as the attack vector, the complexity of the attack, the privileges, the user interaction, the scope, the integrity, and the availability of the data when conducting the attack.

Organizations rely on the CVSS scoring system as a standard way to capture the main features of vulnerabilities and produce a numerical score that reflects their severity.

The calculation of CVSS is modeled according to the risk formula, where the impact is multiplied by the probability, while CVSS calls it the submetric of impact and exploitability: $CVSS = \text{Impact} \times \text{Exploitability}$.

In this image, we can see attack vectors in CVE-2014-9357:

Figure 7.6: Attack vectors in CVE-2014-9357

For obtaining the level of criticality of vulnerabilities, we can also use some metrics such as the access vector, the network, the complexity of access, authentication, and the impact of confidentiality. Among the main attack vectors, we can highlight:

Access vector: This metric reflects how the vulnerability is exploited. An exploitable vulnerability with access to the network means that the vulnerable software is linked to the network stack, and the attacker does not require access to the local network. Such vulnerability is often called

exploitable remotely. An example of a network attack is an RPC buffer overflow.

Buffer overflow is a common vulnerability in web servers that occurs when an application tries to place more data in a buffer that was designed to retain. In the case of a buffer overflow, a programmer creates a buffer in the code but does not place restrictions on it. The data must go somewhere, which in this case, means adjacent buffers. When data overflows in buffers, the result may be corrupt or overwritten data.

Access complexity: This metric analyzes the complexity of the attack required to exploit the vulnerability once an attacker gains access to the system. For example, consider a buffer overflow in Internet service, once the target system is located, the attacker could initiate an exploitation process.

Authentication: This metric analyzes the number of times an attacker must authenticate on a target to exploit a vulnerability. This metric measures that an attacker is required to provide credentials before the vulnerability occurs. The fewer authentication instances required, the higher the vulnerability score.

Confidentiality impact: This metric measures the impact of the confidentiality of a successfully exploited vulnerability. Confidentiality refers to limiting access and disclosure of information to authorized users only, as well as preventing access by unauthorized persons. The greater the impact of confidentiality increases the vulnerability score.

Integrity impact: This metric measures the impact on the integrity of a successfully exploited vulnerability. The greater the impact on integrity increases the vulnerability score. For example, if an attacker can modify any file in the target system, at this point, we have a very high score.

Another vulnerability that affects Docker container with root permissions is one discovered in runc, the utility to run containers of the open containers initiative, by which it is possible to obtain root permissions on the host machine that is running the container.

For the exploitation of the vulnerability, only one malicious container is necessary, which will overwrite the binary runc of the host machine. The attack is not blocked by AppArmor's default policy, just like SELinux's on systems like Fedora. However, using the enforcing mode with a correct configuration of the namespaces (as happens by default in Red Hat), the attack is blocked.

The vulnerability has been identified with CVE-2019-5736, and there is already a patch available that corrects the bug. If it is not possible to patch, a possible solution is to configure SELinux correctly to prevent exploitation.

More information about the vulnerability in <https://seclists.org/oss-sec/2019/q1/119>.

Dirty Cow Exploit (CVE-2016-5195)

DirtyCow (CVE-2016-5195) is a privilege escalation vulnerability in the Linux kernel, and it allows any existing user without privileges to perform an elevation of administrator privileges. It exploits a race condition between processes to enter the kernel and modify files.

COW (Change on Write) is a technique used in UNIX systems to reduce duplication of objects in memory. When using the Race condition, the user with minimal privileges will modify the read-only objects, which within the ideal case should not occur. The main recommendation is to verify the version of your kernel and update it if it detects a vulnerable version.

The vulnerability used in Dirty Cow is a vulnerability that exploits the contents of the memory while the kernel is executing system calls (syscalls) to perform actions in the same memory address space.

The vulnerability opens a file that only the root user with read-only permissions has access to and tries to write some content to the file. Normally this is denied by the privilege hierarchy, but the exploit allows opening the file in a read-only segment in memory.

In this image, you can see the versions that are vulnerable along with the version that would solve the bug:

Vulnerable and fixed packages

The table below lists information on source packages.

Source Package	Release	Version	Status
linux (PTS)	jessie	3.16.56-1+deb8u1	fixed
	jessie (security)	3.16.57-2	fixed
	stretch	4.9.110-1	fixed
	stretch (security)	4.9.110-3+deb9u5	fixed
	buster	4.18.6-1	fixed
	sid	4.18.8-1	fixed

The information below is based on the following data on fixed versions.

Package	Type	Release	Fixed Version	Urgency	Origin	Debian Bugs
linux	source	(unstable)	4.7.8-1	high		
linux	source	jessie	3.16.36-1+deb8u2	high	DSA-3696-1	
linux	source	wheezy	3.2.82-1	high	DLA-670-1	

Figure 7.7: Linux versions affected by DirtyCow

More information on <https://security-tracker.debian.org/tracker/CVE-2016-5195>.

In the following GitHub repositories, you can find some proofs of concept that allow simulating the behavior of this exploit.

<https://github.com/scumjr/dirtycow-vdso>

<https://github.com/gebl/dirtycow-docker-vdso>

In the following GitHub repository <https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs> we can find several proofs of concept where we have the exploit in the dirtycow.c file.

Link	Usage	Description	Family
dirtycow.c	<code>./dirtycow file content</code>	Read-only write	/proc/self/mem
cowroot.c	<code>./cowroot</code>	SUID-based root	/proc/self/mem
dirtycow-mem.c	<code>./dirtycow-mem</code>	libc-based root	/proc/self/mem
pokemon.c	<code>./d file content</code>	Read-only write	PTRACE_POKEDATA
dirtycow.cr	<code>dirtycow --target --string -- offset</code>	Read-only write	/proc/self/mem
dirtycow.c	<code>./dirtycow file content</code>	Read-only write (Android)	/proc/self/mem

Figure 7.8: Exploit files for DirtyCow

In this repository you already can find the Dockerfile and the scripts to run it:

<https://github.com/Alpha-Cybersecurity/dirtyc0w-docker>

Dockerfile	Minor changes (docker -> x86)	5 months ago
README.md	README.md	5 months ago
dirtycow.c	README.md	5 months ago
run.sh	README.md	5 months ago
safe_run.sh	README.md	5 months ago

Figure 7.9: Repository for DirtyCow proof of concept

This is the content of the Dockerfile we can find in the previous repository:

```

1 FROM ubuntu:12.04
2
3 RUN apt-get update
4 RUN apt-get install -y build-essential
5
6 RUN mkdir /dirtycow
7 COPY dirtycow.c /dirtycow/dirtycow.c
8
9 RUN groupadd -r dcow && useradd --no-log-init -r -g dcow dcow
10 RUN echo 'dcow:pass' | chpasswd
11
12 RUN chown -R dcow:dcow /dirtycow
13
14 USER dcow
15
16 WORKDIR /dirtycow
17
18 RUN gcc -pthread dirtycow.c -o dirtycow

```

Figure 7.10: Dockerfile for DirtyCow proof of concept

The DirtyCow environment is based on an Ubuntu image. GCC compiler and build-essential packages are a prerequisite for the compilation of DirtyCow exploit.

In this screenshot, we can see the execution of DirtyCow exploit:

Figure 7.11: Executing DirtyCow proof of concept

The DirtyCow exploit demonstrates how to write to files as a root user (system administrator). In this example, a file called foo is created with certain content, permission, and the property is set, and we try to write to the file as a normal user, but with the DirtyCow binary or exploit.

The vulnerability occurs when opening a file that only the root user has access to read-only permissions and tries to write some content to the file. Normally this is rejected by the privilege hierarchy, but the exploit opens the file in a read-only memory segment.

Prevent DirtyCow with apparmor

If we use AppArmor, we can make the exploit have no effect by establishing restrictions on which applications within the container have permission to read, write, and execute. When using the default apparmor profile, the DirtyCow exploit was stopped. Another possibility is to run containers in read-only mode. The execution of the containers as read-only could prevent an attacker from making changes to the system.

In this screenshot, we can see the execution of DirtyCow exploit with apparmor enabled:

Figure 7.12: Executing DirtyCow with apparmor enabled

AppArmor is a security feature that is part of the Linux kernel and is a tool to restrict the capabilities of an application during runtime.

Another possibility is to run containers in read-only mode. The execution of the containers as read-only could prevent the attacker from making changes to the system.

Vulnerability jack in the box (CVE-2018-8115)

This is a remote code execution vulnerability (CVE 2018-8115) that affects Docker for Windows. This vulnerability is related to the compatibility of Windows Compute Service Shim published and maintained by Microsoft. This service uses a `filepath.Join` function with the unauthorized entry that allows you to create, delete, and replace files on the host's file system, which leads to remote code execution.

The vulnerability is due to the fact that the file path in that function is not validated correctly, and the destination file can be written to an arbitrary location on the victim's host.

The good news is that Docker patched this vulnerability in the Docker CE 18.03.1 and Docker CE 17.05.0-rc1 versions.

Companies like Aqua Security have also created open-source tools that allow you to check images to see if they contain this vulnerability. To do this, the tool downloads an image from the registry, obtains the image layers, and performs a verification of the `.tar` file for each layer. The script can be found in the repository:

<https://github.com/aquasecurity/scancve-2018-8115>

It is a python script that will connect to the Docker Hub Registry and verify if an image, in any of its layers, has any access level path related to the file system that may exploit the CVE-2018-8115 vulnerability.

In this screenshot we can see a fragment of the Python script for detecting CVE-2018-8115 vulnerability:

```

165  def is_layer_safe(layer_file):
166      results = []
167      try:
168          tar = tarfile.open(layer_file, mode='r:gz')
169      except tarfile.ReadError:
170          tar = tarfile.open(layer_file, mode='r')
171
172      while True:
173          next_block = tar.next()
174          if not next_block:
175              break
176
177          filename = next_block.name
178          link_destination = next_block.linkname
179
180          if not os.path.relpath(filename).find('..\\"') or not os.path.relpath(filename).find('..\\'):
181              results.append((filename, 0, layer_file))
182
183          if link_destination:
184              if link_destination[0] not in [ '/', '\\\\' ]:
185                  full_path = os.path.dirname(filename) + "/" + link_destination
186                  if not os.path.relpath(full_path).find('..\\"') or not os.path.relpath(full_path).find('..\\'):
187                      results.append((full_path, 1, layer_file))
188
189

```

Figure 7.13: Python script for detecting CVE-2018-8115 vulnerability

In this section, we have reviewed some vulnerabilities and attacks that are common in some Docker images.

Most vulnerable packages

Finally, we get what packages make Docker images contain vulnerabilities more frequently. For official images, glibc is the most frequently used library, affecting more than 80% of the images in all versions and the latest version.

The following table represents 10 main packages that make images contain vulnerabilities:

Rank	Package name (Percentage of impacted images)			
	Official	Official :latest	Community	Community :latest
1	glibc (89.81%)	glibc (81.91%)	glibc (84.24%)	glibc (84.82%)
2	util-linux (89.55%)	util-linux (81.91%)	openssl (78.32%)	openssl (78.51%)
3	shadow (89.55%)	shadow (81.91%)	util-linux (77.01%)	util-linux (77.24%)
4	perl (87.29%)	audit (77.66%)	shadow (77.01%)	shadow (77.24%)
5	apt (83.82%)	perl (73.40%)	perl (74.07%)	perl (73.05%)
6	openssl (83.79%)	tar (72.34%)	pam (70.92%)	pam (70.53%)
7	tar (83.58%)	apt (70.21%)	pcre3 (66.54%)	audit (67.10%)
8	openldap (76.85%)	openssl (67.02%)	audit (65.48%)	pcre3 (65.59%)
9	krb5 (76.06%)	systemd (67.02%)	krb5 (64.99%)	dpkg (64.36%)
10	audit (73.51%)	gcc (65.96%)	libidn (64.54%)	libidn (62.93%)

Figure 7.14: Most vulnerable packages

As we can see in the previous table, certain packages appear in more than one version image. These packages could be specifically targeted to improve the security of the Docker hub ecosystem.

Analyzing vulnerabilities in Docker images

An audit process ensures that all containers are based on updated containers and that both hosts and containers are configured securely. Among the characteristics that we can validate in an audit process, we can highlight:

Isolation and minimum privilege: the containers are executed with the minimum resources and privileges necessary to function effectively. For this, it is important to limit both the memory and the use of CPU and network functions.

Limiting the amount of memory available to a container will prevent attackers from consuming all the memory on the host and killing other services. Limiting the use of the CPU and the network can prevent attackers from executing denial of service attacks and heavy resource processes such as Bitcoins mining.

Access controls: Linux security modules, such as Apparmor or SELinux, are used to enforce access controls and limit system calls.

Some specific considerations to consider in an audit process:

Check that the images and packages in them are updated and without vulnerabilities.

Links and volumes between containers. Using base file systems in read-only mode will make it easy to find problems with the docker diff command. It is important that our images take up as little space as possible. The larger the images, the more difficult the audit will be.

The kernel of the machine where the docker server is running should always be updated as it is the point shared between all the containers running on the same server.

The NVD database, which is managed by the U.S. government, details the effects for each CVE, including the code that causes it and the correct patch.

https://nvd.nist.gov/vuln/search/results?formtype=Basic&resultstype=overview&query=docker&search_type=all

The screenshot shows the NVD search interface. At the top, there's a blue header with the 'Information Technology Laboratory' logo and the 'NATIONAL VULNERABILITY DATABASE' text. Below the header, there are two tabs: 'VULNERABILITIES' (which is selected) and 'SEARCH AND STATISTICS'. The main content area has a title 'Q Search Results (Refine Search)' and a summary message: 'There are 57 matching records. Displaying matches 1 through 20.' To the right of this message is a 'Sort results by:' dropdown set to 'Publish Date Descending' with a 'Sort' button next to it. Below the summary are three filter buttons: 'Vuln ID ⓘ', 'Summary ⓘ', and 'CVSS Severity ⓘ'. The main table area is currently empty, showing only the column headers.

Figure 7.15: National vulnerability database

NVD assigns to each vulnerability a score of 0 to 10. Range 7-10 scores were graded as a highly critical vulnerability, range 4-6 scores as moderate vulnerability, and 0-4 as low vulnerability.

Vuln ID	Summary	CVSS Severity
CVE-2018-15664	In Docker through 18.06.1-ce-rc2, the API endpoints behind the 'docker cp' command are vulnerable to a symlink-exchange attack with Directory Traversal, giving attackers arbitrary read-write access to the host filesystem with root privileges, because daemon/archive.go does not do archive operations on a frozen filesystem (or from within a chroot).	V3: 8.7 HIGH V2: 9.3 HIGH
	Published: May 23, 2019; 10:29:07 AM -04:00	
CVE-2019-5021	Versions of the Official Alpine Linux Docker images (since v3.3) contain a NULL password for the 'root' user. This vulnerability appears to be the result of a regression introduced in December of 2015. Due to the nature of this issue, systems deployed using affected versions of the Alpine Linux container which utilize Linux PAM, or some other mechanism which uses the system shadow file as an authentication database, may accept a NULL password for the 'root' user.	V3: 9.8 CRITICAL V2: 10.0 HIGH
	Published: May 08, 2019; 01:29:01 PM -04:00	
CVE-2019-1003065	Jenkins CloudShare Docker-Machine Plugin stores credentials unencrypted in its global configuration file on the Jenkins master where they can be viewed by users with access to the master file system.	V3: 8.8 HIGH V2: 4.0 MEDIUM
	Published: April 04, 2019; 12:29:00 PM -04:00	

Figure 7.16: NVD vulnerabilities

This classification takes into consideration several factors, including the complexity needed to exploit a system and vulnerability impact. Lower complexity implies a higher score, and greater impact implies a higher score.

Security vulnerability classification

MITRE is an agency that provides and maintains a CVE (Common Vulnerabilities and Exposures) list of vulnerabilities contained in operating systems and servers. The NVD database, managed by the U.S. government, details the effects for each vulnerability, including its affected code and possible solutions.

For each vulnerability, NVD gives a score of 0 to 10. The range scores 7-10 are rated as high vulnerability, the range scores 4-6 as medium vulnerability, and 0-3.9 as low vulnerability.

Some examples of vulnerabilities classified by level criticality that we can find in docker images are:

High criticality vulnerabilities:

ShellShock [http://en.wikipedia.org/wiki/Shellshock_\(softwarebug\)](http://en.wikipedia.org/wiki/Shellshock_(software_bug))

Heartbleed (OpenSSL) <http://heartbleed.com>

Vulnerabilities of medium criticality:

Poodle (OpenSSL) <https://poodlebleed.com>

Vulnerabilities of low criticality:

Buffer Overflow: GCC memory allocations can cause an overflow buffer due to memory overflow when accessing memory areas that have not been assigned.

To obtain the latest known vulnerabilities of NVD, we have a script developed in Python in the following GitHub repository

<https://github.com/linxack/nvdparser>

The classification of a vulnerability is often subjective, and companies usually classify them depending on specific configurations or the score given by certain Linux distributions. For example, we can take as a reference to the score assigned by a given distribution. For example, for Ubuntu, we have the following list:

<http://people.canonical.com/~ubuntu-security/cve/main.html>

CVE	Package	Ubuntu 12.04 ESM (Precise Pangolin)	Ubuntu 14.04 ESM (Trusty Tahr)	Ubuntu 16.04 LTS (Xenial Xerus)	Ubuntu 18.04 LTS (Bionic Beaver)	Ubuntu 18.10 (Cosmic Cuttlefish)	Ubuntu 19.04 (Disco Dingo)	Ubuntu 19.10 (Eoan)	Links
CVE-2002-2439	gcc-4.6	needs-triage*	DNE	DNE	DNE	DNE	DNE	DNE	Mitre LP Debian
CVE-2005-4890	shadow	needed*	not-affected*	not-affected*	not-affected*	not-affected*	not-affected*	not-affected*	Mitre LP Debian
CVE-2008-7320	seahorse	DNE	DNE	needs-triage*	needs-triage*	needs-triage*	needs-triage*	needs-triage*	Mitre LP Debian
CVE-2009-1384	libpam-krbs5	needed*	needed*	needed*	needed*	needed*	needed*	needed*	Mitre LP Debian
CVE-2009-5080	groff	needed*	needed*	needed*	needed*	needed*	needed*	needed*	Mitre LP Debian
CVE-2009-5155	eglibc	needed*	needed*	DNE	DNE	DNE	DNE	DNE	Mitre LP Debian
CVE-2009-5155	glibc	DNE	DNE	needed*	needed*	not affected*	not affected*	not affected*	Mitre LP Debian

Figure 7.17: Ubuntu CVE list

The red hat also manages its own CVE list available in <https://access.redhat.com/security/security-updates/#/cve>

The screenshot shows the Red Hat Customer Portal interface. At the top, there's a navigation bar with links for 'CUSTOMER PORTAL', 'Products & Services', 'Tools', 'Security', and 'Community'. Below the navigation, a breadcrumb trail reads 'Security Updates > Red Hat CVE Database'. The main content area has three tabs: 'Security Advisories' (gray), 'Red Hat CVE Database' (dark blue, currently selected), and 'Security Labs' (gray). Below these tabs is a search bar with 'Keyword' and 'GO' buttons, and filters for 'All', 'Low', 'Moderate', 'Important', and 'Critical' severity levels. To the right is a 'Filter By Year' dropdown set to 'All'. The main table displays two CVE entries:

CVE	Synopsis	Impact	Publish Date
CVE-2019-10156	** RESERVED ** This candidate has been reserved by an organization or individual that will use it when announcing a new security problem. When the candidate has been publicized, the details for this candidate will be provided.	Moderate	04 Jun 2019
CVE-2019-10149	A flaw was found in Exim versions 4.87 to 4.91 (inclusive). Improper validation of recipient address in deliver_message() function in /src/deliver.c may lead to remote	Critical	04 Jun 2019

Figure 7.18: Red Hat CVE list

In this section, we have reviewed some vulnerabilities classified by the criticality level.

Alpine image vulnerability

This is one of the latest vulnerabilities discovered. The problem is in the alpine Linux image that is distributed for versions 3.3, 3.4, 3.5. These versions contain an empty password for the root user. In addition to exploiting the vulnerability, you need to have the Linux-pam authentication package or the shadow package installed on Linux.

<https://launchpad.net/ubuntu/+source/shadow>

<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5021>

In the following screenshot, we can see the details of this vulnerability:

CVE-ID**CVE-2019-5021**[Learn more at National Vulnerability Database \(NVD\)](#)

• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Versions of the Official Alpine Linux Docker images (since v3.3) contain a NULL password for the 'root' user. This vulnerability appears to be the result of a regression introduced in December of 2015. Due to the nature of this issue, systems deployed using affected versions of the Alpine Linux container which utilize Linux PAM, or some other mechanism which uses the system shadow file as an authentication database, may accept a NULL password for the 'root' user.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- BID:108288
- URL:<http://www.securityfocus.com/bid/108288>
- CONFIRM:<https://security.netapp.com/advisory/ntap-20190510-0001/>
- MISC:<https://alpinelinux.org/posts/Docker-image-vulnerability-CVE-2019-5021.html>
- MISC:https://talosintelligence.com/vulnerability_reports/TALOS-2019-0782
- SUSE:openSUSE-SU-2019:1495
- URL:<http://lists.opensuse.org/opensuse-security-announce/2019-06/msg00004.html>

Figure 7.19: Alpine image vulnerability

In the GitHub repository we can see the change made by maintainers in the alpine image in one of the scripts:

<https://git.alpinelinux.org/aports/commit/?id=7a2566ec8260ceacae81088ebe2ffe6526c3809e>

In the following screenshot we can see script affected for this vulnerability:

Diffstat

-rwxr-Xr-X **scripts/genrootfs.sh** 3

1 files changed, 3 insertions, 0 deletions

```
diff --git a/scripts/genrootfs.sh b/scripts/genrootfs.sh
index ac760e6e0d..5118027632 100755
--- a/scripts/genrootfs.sh
+++ b/scripts/genrootfs.sh
@@ -39,6 +39,9 @@ ${APK:-apk} fetch --keys-dir "$keys_dir" --no-cache \
--repositories-file "$repositories_file" \
--stdout --quiet alpine-base | tar -zx -C "$tmp" etc/
 

+# make sure root login is disabled
+sed -i -e 's/^root:::/root:!:/' "$tmp"/etc/shadow
+
branch=edge
VERSION_ID=$(awk -F= '$1=="VERSION_ID" {print $2}' "$tmp"/etc/os-release)
case $VERSION_ID in
```

Figure 7.20: Script affected in Alpine image vulnerability

With the following command, we can check the 3.4 version is vulnerable since the root user is enabled.

```
$ docker run docker.io/alpine:3.4 cat /etc/shadow | head -n1
```

In the following screenshot we can see the output of the previous command:

Figure 7.21: Docker container running version vulnerable

The solution to this problem is to disable root login with the instruction:

```
RUN sed -i -e 's / ^root:/ root:!:/' / etc/shadow
```

Where the character! means the root user cannot log in. With the following command, we can check the latest version is not vulnerable since the root user is disabled:

```
$ docker run docker.io/alpine:latest cat /etc/shadow | head -n1
```

In the following screenshot we can see the output of the previous command:



Figure 7.22: Docker container running version not vulnerable

In this section, we have reviewed a specific vulnerability in alpine Docker images and the mitigation for this.

CVE in Docker images

We can find CVEs that are directly related to Docker security incidents or issues. To learn more about Docker CVEs or see a list of current Docker CVEs, visit <https://www.docker.com/docker-cve-database>.

In the previous URL, we can find that CVE that are directly related to incidents or security problems in Docker:

CVE ID	Description	Date	Patch
CVE-2016-8867	Incorrect application of ambient capabilities	Oct 27, 2016	Engine 1.12.3
CVE-2014-8178	Attacker controlled layer IDs lead to local graph content poisoning	Oct 12, 2015	Engine 1.8.3, 1.6.2-CS7
CVE-2014-8179	Manifest validation and parsing logic errors allow pull-by-digest validation bypass	Oct 12, 2015	Engine 1.8.3, 1.6.2-CS7
CVE-2015-3629	Symlink traversal on container respawn allows local privilege escalation	May 7, 2015	Engine 1.6.1
CVE-2015-3627	Insecure opening of file-descriptor 1 leading to privilege escalation	May 7, 2015	Engine 1.6.1
CVE-2015-3630	Read/write proc paths allow host modification & information disclosure	May 7, 2015	Engine 1.6.1
CVE-2015-3631	Volume mounts allow LSM profile escalation	May 7, 2015	Engine 1.6.1

Figure 7.23: Docker CVE database

This list will be updated every time a CVE is detected for a specific version of Docker. As you can see, the list is very small, so it's probably a list that won't grow on a basic frequency, day-to-day, or even month-to-month.

We can find other services that allow us to visualize the Docker vulnerabilities registered in the CVE database. <https://www.saucs.com/cve?vendor=docker>.

In the following screenshot we can see some CVE registered in saucs database:

CVE	Vendors	Products	Updated	CVSS
CVE-2019-14271	1 Docker	1 Docker	2019-08-28	7.5
		In Docker 19.03.x before 19.03.1 linked against the GNU C Library (aka glibc), code injection can occur when the nsswitch facility dynamically loads a library inside a chroot that contains the contents of the container.		
CVE-2019-13509	1 Docker	1 Docker	2019-08-27	5.0
		In Docker CE and EE before 18.09.8 (as well as Docker EE before 17.06.2-ee-23 and 18.x before 18.03.1-ee-10), Docker Engine in debug mode may sometimes add secrets to the debug log. This applies to a scenario where docker stack deploy is run to...		
CVE-2019-1020014	1 Docker	1 Credential Helpers	2019-08-19	2.1
		docker-credential-helpers before 0.6.3 has a double free in the List functions.		
CVE-2018-15664	1 Docker	1 Docker	2019-06-25	6.2
		In Docker through 18.06.1-ce-rc2, the API endpoints behind the 'docker cp' command are vulnerable to a symlink-exchange attack with Directory Traversal, giving attackers arbitrary read-write access to the host filesystem with root privileges,...		
CVE-2019-5736	10 Docker, Google, Linuxcontainers and 7 more	12 Docker, Kubernetes Engine, Lxc and 9 more	2019-06-03	9.3

Figure 7.24: Docker CVE sauc database

As an entity handling and publishing CVEs, MITRE correlates every CVE with the program that is affected by this vulnerability. <https://cve.mitre.org>

Next, we will study some of these vulnerabilities, related to specific versions of Docker:

CVE-2014-5282: Docker prior to version 1.3 does not correctly validate image identifiers, allowing remote attackers to redirect to another image by loading untrusted images using docker load: <https://nvd.nist.gov/vuln/detail/CVE-2014-5282>

CVE-2014-5280: boot2docker prior to version 1.3 allows attackers to perform Cross-Site Request Forgery (CSRF) attacks by taking advantage of Docker daemons by enabling TCP connections without TLS encryption: <https://nvd.nist.gov/vuln/detail/CVE-2014-5280>

CVE-2014-5279: The boot2docker controlled Docker Daemon before 1.3 makes improper TCP connections without default authentication, making it easier for a remote attacker to gain permissions or execute arbitrary code in containers: <https://nvd.nist.gov/vuln/detail/CVE-2014-5279>

When a new vulnerability is discovered, creating a fresh CVE is a way to make it accessible to the public. This CVE includes all the vulnerability data, as well as an identifier unique for each security vulnerability identified.

Vulnerable images in Docker hub

In the next URL in the Docker hub, we can find some images that already bring by default a series of vulnerabilities.

<https://hub.docker.com/u/vulnerables>

Vulnerable ShellShock image

<https://hub.docker.com/r/vulnerables/cve-2014-6271>

Shellshock is a family of security errors in the bash on Unix systems. Many web server implementations use bash to process requests, allowing an attacker to execute arbitrary commands and gain unauthorized access to a computer system.

Image with OpenSSH remote DOS vulnerability

<https://hub.docker.com/r/vulnerables/cve-2016-6515>

Versions prior to 7.3 of OpenSSH do not limit the length of passwords for authentication on SSH servers, which allows a remote attacker to cause a denial of service over a long chain. This error resides in the source code of the auth-passwd.c file in the auth_password function.

An attacker can take advantage of this problem to make the application go into an infinite loop and consume CPU resources until it causes a denial of service.

Vulnerable HeartBleed Image

<https://hub.docker.com/r/vulnerables/cve-2014-0160>

The Heartbleed error is a vulnerability in the OpenSSL cryptographic software library. This vulnerability compromises the secret keys used to identify service providers and encrypt user traffic, names, and passwords. It is thought that about 17% (half a million) of secure internet web servers certified by trusted authorities are vulnerable to attack, allowing theft of private keys and session passwords and cookies from servers.

In the following services, we can find containers with latest updates about vulnerabilities:

<https://vulnerablecontainers.org/>

<https://vulnerablecontainers.org/official/>

At this point, we have reviewed some main CVE we can find in Docker images. Next section, we will review how we can get CVE details with vulners API.

Getting CVE details with vulners API

Vulnersdatabase (<https://vulners.com/products>) provides searches, data recovery, archiving, and vulnerability scanning API for integration purposes. With this library, it is possible to create security tools and access the largest security database in the world.

It provides an API in Python to obtain information about the CVE by identifier, search for available public vulnerabilities and obtain vulnerabilities and vulnerabilities by name and software version.

<https://github.com/vulnersCom/api/blob/master/README.md>

To install the library, just run the command pip install:

```
$ pip install -U vulners
```

In addition, it is necessary to register on the vulners website to obtain "API KEY" that allows you to make requests and queries. Below are some examples of queries to this database.

The following Python script allows you to search in the vulners database by specific search criteria; for example, we could search for the DirtyCow vulnerability.

```
import vulners

vulnersapi = vulners.Vulners(apikey="API_KEY")

dirtycow = vulners_api.search("dirtycow", limit=10)

for i, val in enumerate(dirtycow):

    for key,value in val.items():

        print(key,":",value)
```

In the following screenshot you can see the output of the previous script:

Figure 7.25: Information about DirtyCow

We could also obtain more information about a specific CVE.

In the following code we are using the document method from the vulners API for searching a specific CVE identifier:

```
vulnersapi = vulners.Vulners(apikey="API_KEY")  
CVE20188115 = vulners_api.document("CVE-2018-8115")  
print(type(CVE20188115))  
  
for key,value in CVE20188115.items():  
    print(key,":",value)
```

In the following screenshot you can see the output of the previous script:

Figure 7.26: Information about CVE-2018-8115

We could obtain information about public exploits with the searchExploit method.

```
import vulners  
  
vulnersapi = vulners.Vulners(apikey="API_KEY")  
  
wordpressexploits = vulnersapi.searchExploit("wordpress 4.7.0")
```

Get references for a specific CVE identifier.

```
import vulners  
  
vulnersapi = vulners.Vulners(apikey="API_KEY")  
  
references = vulners_api.references("CVE-2018-8115")  
  
for key,value in references.items():  
    for key,val in enumerate(value):  
        for key,value in val.items():  
            print(key,":",value)
```

In this section we have reviewed vulner API for get more information about vulnerabilities and attacks that are common in some Docker images. You can get more information about the API and more examples in Git repository <https://github.com/vulnersCom/api>

Conclusion

In this chapter we have reviewed some topics like Docker container threats and examples of container attacks like DirtyCow. Also, we have reviewed specific CVE in Docker images and how we can get more information about specific vulnerability with vulners API.

In this chapter, the reader has learned about the main vulnerabilities and Docker threats we can find in Docker images and containers. As a result, developers will have the capacity to obtain details about specific CVEs in container applications.

In the next chapter, we will review the state of Kubernetes security, and some tools for checking Kubernetes is implemented in a secure way by following some best practices documented in the CIS Kubernetes Benchmark guide.

Questions

Which is the metric being used to measure and compare threats and vulnerabilities?

Which common vulnerability in web servers occurs when an application tries to place more data in a buffer that was designed to retain?

Which vulnerability affects Docker container with root permissions discovered in a utility to run containers of the open containers initiative?

Which vulnerability exploits the contents of the memory while the kernel is executing system calls (syscalls) to perform actions in the same memory address space?

Which package is the most frequently used in Docker images and is one of the most vulnerable libraries?

CHAPTER 8

Kubernetes Security

In this chapter, we will introduce Kubernetes security and Kubernetes Bench for security project as an application that checks whether Kubernetes is implemented securely by executing the controls documented in CIS Kubernetes Benchmark guide. Also, we will review more critical vulnerabilities discovered in the last year and other security projects to improve security in Kubernetes.

In this chapter, the reader will learn about the state of the Kubernetes security and what are the main tools we can find in the Kubernetes ecosystem for checking the security of the Kubernetes cluster. As a result, DevOps will have the capacity to analyze the security and risks that containers and pods are exposing to Docker and Kubernetes clients.

Structure

Introducing Kubernetes security

Kubernetes engine security

KubeBench security and vulnerabilities

Kubernetes security projects

Objectives

Understanding the principles and best practices about Kuberne's security.

Knowing about security risks in Kubernetes.

Knowing about KubeBench security and main vulnerabilities affected in the last year.

Knowing about Kubernetes security projects and plugins for testing the security of your Kubernetes cluster.

Introducing Kubernetes security

If you don't want to install anything on your local machine, you can play with Kubernetes with an online service that allows you to have four-hour environments, totally free, where you can create a cluster with several nodes in a fast way.

In the following links, you will find some resources related to executing Kubernetes online, and you could play with some scenarios that are configured in the online environment:

<https://labs.play-with-k8s.com/>

<https://training.play-with-kubernetes.com/>

<https://github.com/play-with-docker/play-with-kubernetes.github.io>

Securing containers with Kubernetes

Kubernetes and Docker are revolutionizing the world of computing, application development, and specifically DevSecOps. Both technologies combined offer us benefits such as scaling and managing the implementation of an application or a service by using containers, to the point of becoming today a true standard for orchestration. Like any other infrastructure, we must take precautions at the time of its implementation to try to build it as safely as possible, as well as offer the best final performance.

From the perspective of DevOps, Kubernetes brings the following characteristics:

Operating in the DevOps model: In the DevOps model, software developers assume greater responsibility for building and deploying applications.

Creation of common service sets: Actually, applications request a service from another application pointing to an IP address and port number. With Kubernetes, it is possible to build applications in containers that provide services that are available for other containers to use.

Data-center pre-configuration: Kubernetes aims to create consistent application programming interfaces (APIs) that result in stable environments for running applications in containers. Developers should be able to create applications that work in any cloud provider that supports those APIs. This reliable framework means that developers can identify the version of Kubernetes along with the services they need and not have to worry about the specific configuration of the data-center.

Configuring Kubernetes

While Docker manages entities referred to like images and containers, Kubernetes wraps those entities in what is referred to as pods. A pod can contain one or more running containers and is the unit that manages Kubernetes. There are several advantages that Kubernetes brings to container management as pods:

Multiple nodes: Instead of simply deploying a container on a single host, Kubernetes can implement a set of pods on multiple nodes. Essentially, a node provides the environment where a container is executed.

Replication: Kubernetes can act as a replication controller for a pod. This means that you can set how many replicas of a specific pod should be running at all times.

Services: The word "service" in the context of Kubernetes implies that you can assign a service name (ID) to a specific IP address and port, and then assign a pod to provide that service. Kubernetes internally tracks the location of that service and has the capacity to redirect requests from another pod of that service to the correct address and port.

If you choose to configure Kubernetes, it is important to understand the following concepts before starting:

Kubernetes controller: A Kubernetes controller acts as a node from which the pods, replication controllers, services, and other components of a Kubernetes environment are implemented and managed. To create a Kubernetes controller, you must configure and run the `systemd`, `kube-api-server`, `kube-controller-manager`, and `kube-scheduler` services .

Kubernetes nodes: A Kubernetes node provides the environment in which the containers run. To run a machine as a Kubernetes node, it must be configured to run the `Docker`, `kube-proxy` , and `kubelet` services. These services must be run on each node of the Kubernetes cluster.

`kubectl` command: Most Kubernetes administration is performed on the master node using the `kubectl` command. With `kubectl` , it is possible to create, obtain, describe, or eliminate any of the resources that Kubernetes manages (pods, replication controllers, services, etc.).

Resource files (YAML or JSON): When you create a pod, a replication controller, service, or another resource in Kubernetes, the `kubectl` command expects the information needed to create that resource to be in one of these two types of formats.

The best way to see how Kubernetes works is to configure a Kubernetes cluster that has a master controller node and must have at least two nodes, each operating on separate systems. The Kubernetes API, managed by a `kubelet`, must be protected to ensure that it is not accessed in an unauthorized way to perform malicious actions. In the case that unauthorized access was made to one of the containers running in a pod of a Kubernetes environment, the API could be attacked by means of some simple commands to be able to visualize the information about the entire environment.

Security on Kubernetes should be focused on preventing image manipulation and unauthorized access to the entire environment. Regarding runtime protection, it is essential not to deploy pods with root permissions, checking that pods have defined security policies (pod security policies), and Kubernetes is using secrets for credential and password management.

Best security practices with Kubernetes

From the perspective of security, due to the impact that some implementations that can be carried out in an organization can cause, it is advisable to follow some best practices at a security level. In the following points, we will comment on the main security practices with Kubernetes:

Firewall ports

This security practice is frequently used since it is not advisable to expose a port that does not need to be exposed. In order to prevent this from happening, it is ideal to define the port's exposure:

The first thing you should do is check the existence of some interface or define an IP to link the service, for example, the localhost interface `127.0.0.1` . Some processes are opening so many ports on all interfaces that they should rather have a public access firewall. Although they only allow purely confidential information, they also allow you direct access to your set of computers.

Restrict the Docker pull command

Docker is a resource that can sometimes be uncontrolled by the ease of access it has. That is, anyone with access to the Kubernetes API or Docker connector can obtain the image they want, generating traffic from infected images or with serious security problems for Kubernetes. Even many clusters have already become a network of Bitcoin miners. Although it is a problem that seems not to be solved, the Image Policy plugin can significantly improve that situation, connecting directly with the Docker API. This plugin imposes a series of strict security rules that reflect a black and white list of images that can be extracted.

Another solution is using the Image Policy Webhook through Admission Controller, which intercepts all image extractions and takes care of security in the same way as the plugin mentioned above.

API authorization mode and anonymous authentication

It is very important that you know what authorization mode your system is using. This can be done by verifying the parameters, where you can also check if authentication is configured anonymously.

It is important to know that this configuration will not affect the kubelet authorization mode since it exposes an API on its own that executes commands that kubelet can completely ignore. More specifically, a kubelet provides a command API used by kubernetes , in which arbitrary commands are executed on a specific node. This configuration can be designed in the following way: --authorization-mode = Webhook and --anonymous-auth = false .

Kubernetes dashboard

Kubernetes dashboard offers the option of receiving a service account in which you can view all the operations in your network with full access. The challenge with the Kubernetes dashboard is to restrict public access to it.

You can follow this guide to secure the Kubernetes board:

<https://blog.heptio.com/on-securing-the-kubernetes-dashboard-16b09b1b7aca>

Checking network policies

Network policies represent a series of firewall rules for Kubernetes. Therefore, it is good that you consult the network policies of Kubernetes to configure them correctly from the beginning. See the Kubernetes network policy for an excellent starting point. If your network provider does not support network policies, consider switching to one that supports it.

In the following link, you can find more information about this aspect:

<https://kubernetes.io/docs/concepts/cluster-administration/networking>

Pods security policies

Pods (one or more containers which share network and storage configurations) are the main component of Kubernetes. Therefore, their security is very important and needs to be implemented from the first steps of its design, using security policies. In the official documentation, you can find some examples of how to apply these security policies in our implementation with Kubernetes.

<https://kubernetes.io/docs/concepts/policy/pod-security-policy>

According to official documentation, a pod security policy is a cluster-level resource that controls aspects about the security of a pod. These security policies are defined through the PodSecurityPolicy object, through which we can define the conditions under which a pod must meet to be accepted in the system, and also allows us to define the default values of fields that are not explicitly assigned.

<https://cloud.google.com/kubernetes-engine/docs/how-to/pod-security-policies?hl=es>

A security policy is defined as practically everything in Kubernetes, through a manifest file, usually in YAML format. Let's see an example:

```
apiVersion: policy/v1beta1
```

```
kind: PodSecurityPolicy
```

```
metadata:
```

```
name: permissive
```

```
spec:
```

```
privileged: true  
hostNetwork: true  
hostIPC: true  
hostPID: true  
seLinux:  
rule: RunAsAny  
supplementalGroups:  
rule: RunAsAny  
runAsUser:  
rule: RunAsAny  
fsGroup:  
rule: RunAsAny  
hostPorts:  
• min: 0  
max: 65535  
volumes:  
• /*
```

In this example, you can see how the defined policy is very permissive. It practically allows us to run a pod with all kinds of privileges. For example, we could execute it in privileged mode (privileged: true), so that we could have access to parts of the host, share the space of network names, processes and IPC (Inter-Process Communication) of the host, run the container or containers as root, etc. Unless there is a good reason, such configurations should be avoided.

Pod security policies allow administrators to control the following aspects:

Containers in privileged mode: This feature allows or does not allow the execution of containers in privileged mode. The field that sets this aspect is called privileged. By default, the containers run in non-privileged mode.

Host namespace: There are four fields that allow us to define the behavior of a container with respect to access to certain parts of the host:

HostPID: This controls whether the pod containers share the same process space (IDs) of the host.

HostIPC: This controls whether the containers in a pod share the host's IPC space.

HostNetwork: This controls whether a pod can make use of the same host network space. This implies that the pod would have access to the loopback device, and to the processes that are running on that host.

HostPorts: This defines ranges of ports allowed in the host network space. This range is given by the HostPortRange field, and the min and max attributes that define the range of ports, the values of both attributes are included in the range.

Volumes and file systems:

Volumes: This provides a list of permitted volumes. These correspond to the source used to create the volume.

FSGroup: This controls which supplementary group that applies to some volumes.

AllowedHostPaths: Specifies a list of paths allowed to be used by volumes. An empty list would imply that there are no restrictions. This list is defined by two attributes: pathPrefix and readOnly .

ReadOnlyRootFilesystem: This requires that the containers run with the root file system in read-only mode.

Users and groups:

RunAsUser: Specifies which user the containers run inside the pod.

RunAsGroup: Specifies with which group ID the containers run within the pod.

Privilege escalation: Basically, it controls the `nonewprivs` option of the container process. This option prevents binaries with the `setuid` option from changing the user's effective ID and prevents enabling new extra capabilities.

allowPrivilegeEscalation: Specifies whether or not to set the security context of the container. By default, `allowPrivilegeEscalation = true` to avoid problems with binaries with `setuid` active.

DefaultAllowPrivilegeEscalation: This allows you to set the default option of `allowPrivilegeEscalation`.

Capabilities: GNU/Linux capabilities are a series of superuser privileges, which can be enabled or disabled independently. The following fields accept the capabilities as a list, without the `CAP_` prefix (all capabilities in GNU/Linux begin with that prefix).

AllowedCapabilities: List of capacities that can be added to a container. By default, all capacities are allowed. If this field is specified empty, it implies that you cannot add capacities to a container, beyond those defined by default. The asterisk (*) can be used to refer to all capabilities.

RequiredDropCapabilities: List of capacities that must be removed from the container. These are removed from the default capacity group. The capabilities included in this field should not be included in `AllowedCapabilities` or `DefaultAddCapabilities`.

DefaultAddCapabilities: Capabilities added by default to a default container.

Managing secrets

A secret is everything that nobody else in the cluster should know, neither the rest of the applications nor the operators (or users) that access the cluster. For example, a password from a certificate store, an API key (API Key) so that an application can consume third-party resources, etc.

Let's say that someone discharges those resources along with certain permissions. From there, it is the application that requests those secrets from K8s by presenting the information that authorizes them to consume those resources.

Authorization management is done through what is known as RBAC (role-based access control), that is, only if the application has a certain role can it access certain types of resources. Also, it's important to configure these roles and have released the secret before the application is deployed.

Kubernetes engine security

Kubernetes has become a standard way of implementing applications in containers at scale and helps us handle complex and complex container deployments. As Kubernetes grows and evolves, some of its excesses are likely to be controlled from within. But some people are not expecting Kubernetes to be easier to use, and they have released their own solutions to many common problems with Kubernetes in production.

Handle security risks in Kubernetes

Among the main strategies that we can follow to manage the risks of putting your application with Kubernetes in production, we can highlight:

Integrate security from the early stages of development: With Kubernetes, it is necessary to integrate security at each stage of the software development process. It is a mistake to leave security settings as the last step, as it may be too late.

Consider a commercial platform of Kubernetes: When you participate in a Kubernetes trading platform, the most important benefit you get is the rapid structural responses from development to any threat or problem. Kubernetes will be updated quickly to any vulnerability, and you will always have the latest security updates for your company.

Do not trust your old tools and practices: The attackers update faster than the software, so the same moves at any time may be obsolete. You should not assume that your conventional security tools will protect you. There are many open-source tools that evaluate Kubernetes clusters or perform penetration tests on clusters and nodes. Experts point out that it is necessary to keep a combination of maintaining keep updated and patched your software, for example, and new approaches and tools.

Increasing security using containers with Kubernetes

With the speed of development in Kubernetes, there are often new security features and configurations that you may not know. Next, we are going to study different ways to give more security to a Kubernetes engine cluster.

When running a Kubernetes cluster, there are several best practices to follow.

Among the best security practices for your Kubernetes cluster, we can highlight:

Use the minimum privilege principle for your service accounts.

Disable Kubernetes dashboard.

Disable inherited authorization.

Create a cluster network policy.

Use the minimum privilege principle for your service accounts

The principle of minimum privilege helps reduce the impact of a potential vulnerability or data that has been compromised. Thus, if a certain component is compromised, it will be more difficult for a potential attacker to escalate privileges.

If you are using the Google Cloud Platform, each Kubernetes Engine node has an associated service account. The first thing that should be done is to analyze the accesses that the account has by default and see the permissions that are really necessary to run your Kubernetes cluster.

At this point, it is recommended to use a service account with the minimum privileges to run the Kubernetes Engine Cluster instead of the default service account.

<https://cloud.google.com/monitoring/access-control#overview>

<https://cloud.google.com/logging/docs/access-control#overview>

The following commands will create a GCP service account for you with the minimum permissions necessary to operate Kubernetes engine:

```
$ gcloudiam service-accounts create "${SA_NAME}" \
--display-name="${SA_NAME}"

$ gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
--member "serviceAccount:${SANAME}@${PROJECTID}.iam.gserviceaccount.com" \
--role roles/logging.logWriter

$ gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
--member "serviceAccount:${SANAME}@${PROJECTID}.iam.gserviceaccount.com" \
--role roles/monitoring.metricWriter

$ gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
--member "serviceAccount:${SANAME}@${PROJECTID}.iam.gserviceaccount.com" \
--role roles/monitoring.viewer
```

If you need your Kubernetes engine cluster to have access to other Google Cloud services, we recommend that you create an additional role and supply it to workloads through the Kubernetes secrets. You can do it following the official documentation:

<https://cloud.google.com/kubernetes-engine/docs/tutorials/authenticating-to-cloud-platform>

Disable Kubernetes dashboard

At this point, it's important to know how to disable the Kubernetes web user interface when it runs on Kubernetes Engine. The cloud console provides many of the same features, so you don't need these permissions if you are running the Kubernetes engine.

<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>

The following command disables the Kubernetes web user interface:

```
$ gcloud container clusters update "${CLUSTER_NAME}" --update-addons=KubernetesDashboard=DISABLED
```

Disable inherited authorization

At Kubernetes version 1.8, attribute-based access control (ABAC) is disabled by default in the Kubernetes engine. At this point, a new feature called RBAC has been released in Kubernetes 1.8. RBAC is a new mechanism Kubernetes provide to assign permissions and privileges to roles instead of to specific users.

To create a new cluster with all the previous recommendations, you can execute the following command:

```
$ gcloud container clusters create "${CLUSTER_NAME}" \
--service-account="${SANAME}@${PROJECTID}.iam.gserviceaccount.com"
--no-enable-legacy-authorization \
--disable-addons=KubernetesDashboard
```

Create a cluster network policy

In addition, it is important to create network policies to control the communication between the pods and the services in your cluster. The application of network policies makes it much more difficult for a potential attacker to obtain high privileges within the cluster. We could also use the Kubernetes network policy API to create firewall rules at the pod level in the Kubernetes engine. These firewall rules will determine which pods and services can communicate with each other within the cluster.

<https://cloud.google.com/kubernetes-engine/docs/how-to/network-policy>

To enable the application of network policies when creating a new cluster, you can specify the ---enable-network-policy flag using gcloud command:

```
$ gcloud container clusters create "${CLUSTER_NAME}" \
--project="${PROJECT_ID}" \
--zone="${ZONE}" \
--enable-network-policy
```

KubeBench security and vulnerabilities

KubeBench is a Kubernetes security scanner that allows us to eliminate about 95% of configuration defects, generating quite specific guidelines to ensure the configuration of your computer network through the application of Kubernetes benchmark.

KubeBench is an application made in Golang that checks if Kubernetes is implemented in a secure way by executing controls documented in CIS Kubernetes Benchmark.

CIS Benchmarks for Kubernetes with Kube-bench

CIS Benchmarks are security standards for different systems, carried out by the Center for Internet Security, and which aim to harden our Operating Systems. Compliance with these standards is common in environments that have to meet PCI-DSS, GDPR, or are for government use, so if we are concerned about security, we will always be right if we meet CIS Benchmarks.

To verify the rules of CIS Benchmark, we will use Kube bench, which is an Aqua tool that will automate the entire process of validation of CIS Benchmark rules for Kubernetes.

<https://github.com/aquasecurity/kube-bench>

It is possible to install kube-bench through this dedicated container by executing the following container:

<https://hub.docker.com/r/aquasec/kube-bench>

This tool supports tests for multiple versions of Kubernetes (1.6, 1.7, 1.8, and 1.11) defined in the CIS 1.0.0, 1.1.0, 1.2.0, and 1.3.0 guides, respectively. The easiest way to run this tool is to run it from a container and launch the tests on the Kubernetes cluster with the following command:

```
$ docker run --pid=host -v /etc:/etc:ro -v /var:/var:ro -t
```

aquasec/kube-bench:latest

In this way, we can execute the command for analyzing the master node or a worker node:

Figure 8.1: Kube-bench execution

The tests are configured with YAML and JSON files, making it easy to update this tool as the test specifications evolve. Among the tests, we can perform, we can highlight those related to the parameters `--allow-privileged` and `--anonymous-auth`.

Validating workers

For validating the workers, we can raise a pod that automatically checks each of the rules with the `kubectl` command:

```
$ kubectl run --rm -i -t kube-bench-node --image=aquasec/kube-bench:latest --restart=Never --overrides='{ "apiVersion": "v1", "spec": { "hostPID": true } }' -- node --version 1.8
```

Where `--version` is the version of Kubernetes we are using. The `--overrides` parameter represents what the container is running with the host PID so that it has permissions for checking the host.

In the following screenshot we can see the execution over a worker node:

```

~$ kubectl logs kube-bench-node
[INFO] 2 Worker Node Security Configuration
[INFO] 2.1 Kubelet
[FAIL] 2.1.1 Ensure that the --allow-privileged argument is set to false (Scored)
[PASS] 2.1.2 Ensure that the --anonymous-auth argument is set to false (Scored)
[PASS] 2.1.3 Ensure that the --authorization-mode argument is not set to AlwaysAllow (Scored)
[PASS] 2.1.4 Ensure that the --client-ca-file argument is set as appropriate (Scored)
[PASS] 2.1.5 Ensure that the --read-only-port argument is set to 0 (Scored)
[FAIL] 2.1.6 Ensure that the --streaming-connection-idle-timeout argument is not set to 0 (Scored)
[FAIL] 2.1.7 Ensure that the --protect-kernel-defaults argument is set to true (Scored)
[FAIL] 2.1.8 Ensure that the --make-iptables-util-chains argument is set to true (Scored)
[FAIL] 2.1.9 Ensure that the --keep-terminated-pod-volumes argument is set to false (Scored)
[FAIL] 2.1.10 Ensure that the --hostname-override argument is not set (Scored)
[FAIL] 2.1.11 Ensure that the --event-qps argument is set to 0 (Scored)
[PASS] 2.1.12 Ensure that the --tls-cert-file and --tls-private-key-file arguments are set as appropriate (Scored)
[PASS] 2.1.13 Ensure that the --cadvisor-port argument is set to 0 (Scored)
[FAIL] 2.1.14 Ensure that the RotateKubeletClientCertificate argument is set to true
[FAIL] 2.1.15 Ensure that the RotateKubeletServerCertificate argument is set to true
[INFO] 2.2 Configuration Files
[FAIL] 2.2.1 Ensure that the kubelet.conf file permissions are set to 644 or more restrictive (Scored)
[FAIL] 2.2.2 Ensure that the kubelet.conf file ownership is set to root:root (Scored)
[FAIL] 2.2.3 Ensure that the kubelet service file permissions are set to 644 or more restrictive (Scored)
[FAIL] 2.2.4 2.2.4 Ensure that the kubelet service file ownership is set to root:root (Scored)
[FAIL] 2.2.5 Ensure that the proxy kubeconfig file permissions are set to 644 or more restrictive (Scored)
[FAIL] 2.2.6 Ensure that the proxy kubeconfig file ownership is set to root:root (Scored)
[WARN] 2.2.7 Ensure that the certificate authorities file permissions are set to 644 or more restrictive (Scored)
[WARN] 2.2.8 Ensure that the client certificate authorities file ownership is set to root:root

```

Figure 8.2: Kube-bench execution in a worker node

Validating master

For validating the master node, in the documentation, it is indicated that we must execute a container in the respective node. If we allow our master node to execute container, we can use kubectl command.

```
$ kubectl run --rm -i -t kube-bench-master --image=aquasec/kube-bench:latest --restart=Never --overrides='{"apiVersion": "v1", "spec": { "hostPID": true, "nodeSelector": {"kubernetes.io/role": "master"}, "tolerations": [ {"key": "node-role.kubernetes.io/master", "operator": "Exists", "effect": "NoSchedule"} ] } }' -- master --version 1.8
```

If we do not allow pods to run on the master node, then we can directly execute the container in the master node's Docker with the command:

```
$ docker run --pid=host -t aquasec/kube-bench:latest master --version 1.8
```

In the following screenshot we can see the execution over a master node:

```
[root@master ~]# docker run --pid=host -t aquasec/kube-bench:latest master --version 1.8
[INFO] 1 Master Node Security Configuration
[INFO] 1.1 API Server
[FAIL] 1.1.1 Ensure that the --anonymous-auth argument is set to false (Scored)
[PASS] 1.1.2 Ensure that the --basic-auth-file argument is not set (Scored)
[PASS] 1.1.3 Ensure that the --insecure-allow-any-token argument is not set (Scored)
[PASS] 1.1.4 Ensure that the --kubelet-https argument is set to true (Scored)
[PASS] 1.1.5 Ensure that the --insecure-bind-address argument is not set (Scored)
[PASS] 1.1.6 Ensure that the --insecure-port argument is set to 0 (Scored)
[PASS] 1.1.7 Ensure that the --secure-port argument is not set to 0 (Scored)
[FAIL] 1.1.8 Ensure that the --profiling argument is set to false (Scored)
[FAIL] 1.1.9 Ensure that the --repair-malformed-updates argument is set to false (Scored)
[FAIL] 1.1.10 Ensure that the admission control policy is not set to AlwaysAdmit (Scored)
[FAIL] 1.1.11 Ensure that the admission control policy is set to AlwaysPullImages (Scored)
[FAIL] 1.1.12 Ensure that the admission control policy is set to DenyEscalatingExec (Scored)
[FAIL] 1.1.13 Ensure that the admission control policy is set to SecurityContextDeny (Scored)
[FAIL] 1.1.14 Ensure that the admission control policy is set to NamespaceLifecycle (Scored)
[FAIL] 1.1.15 Ensure that the --audit-log-path argument is set as appropriate (Scored)
[FAIL] 1.1.16 Ensure that the --audit-log-maxage argument is set to 30 or as appropriate (Scored)
```

Figure 8.3: Kube-bench execution in a master node

Kubernetes vulnerabilities

One of the most critical vulnerabilities detected in 2018 has been one we can find in the CVE database with the code CVE-2018-1002105. The vulnerability, considered of critical severity, is in the Kubernetes API server and would allow compromising the pods in execution.

<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1002105>

The vulnerability has been identified in the Kubernetes API server and has been categorized as critical with punctuation CVSS 9.8. The vulnerability allows any authenticated Kubernetes user to obtain administrative access to the cluster using standard security settings and allows the escalation of Kubernetes privileges through a specially designed proxy request.

It is important to note that all Kubernetes-based services and products, including Red Hat products such as OpenShift Container Platform, are affected, so we can also find the reference in the RedHat database:

<https://access.redhat.com/security/cve/cve-2018-1002105>

The vulnerability is due to a vulnerable TCP connection, through which a remote attacker could send specially manipulated requests to one of the added APIs of the Kubernetes API server and escalate privileges using the TLS credentials of that service. The problem is that an unauthenticated user can access the API to create new services that could be used to inject malicious code.

Any user can establish a connection through the Kubernetes API to a server in the backend. Once the connection is established, an attacker can send arbitrary requests directly to that service, and these requests are authenticated with the transport layer security credentials (TLS) of the Kubernetes server.

The bug can be used in two ways: one related to a normal user with exec, attach permissions over a group of containers that share storage and network resources. You could realize privilege escalation at the cluster-admin level and execute any process in a container.

The following command would allow you to discover the APIs added to the cluster:

```
$ kubectl get apiservices -o 'jsonpath= {range.items[?(@.spec.service.name!="")]}{.metadata.name}\n{end}'
```

The RBAC policies, by default, allow these requests to be made to any user, whether or not they are authenticated.

<https://kubernetes.io/docs/reference/access-authn-authz/rbac/#discovery-roles>

Ultimately, an attacker who manages privileges escalation through any of the APIs could access a pod in execution, list the pods in a specific node, and execute arbitrary commands or reveal sensitive information.

The vulnerability has already been corrected by the Kubernetes development team, and it is recommended to update it with patched versions. The only solution for this vulnerability is to update Kubernetes, specifically to the patched version of Kubernetes v1.10.11, v1.11.5, v1.12.3, and v1.13.0-rc.1.

More information in <https://github.com/kubernetes/kubernetes/issues/71411> .

Kubernetes security projects

In this section, we will review different security projects that can help us, both to secure our Kubernetes cluster and to offer the best possible performance to our infrastructure.

Kube-hunter

Kube-hunter is a python script developed by Aqua Security that allows analyzing the potential vulnerabilities in a Kubernetes Cluster. In this example, we are using the network scanning mode to internally review all the nodes of the cluster, although this tool has many more options such as active hunting or container/pod deployment.

<https://github.com/aquasecurity/kube-hunter>

Kubernetes clusters are mounted on a set of nodes or servers in which at least one of them has to take the role of master, and the rest are defined as workers. These nodes have visibility with each other in order to communicate. It is important to know that the main ports for the management of these clusters are: 443, 8080, 10250 , and 10255 .

This tool allows you to perform a security vulnerability analysis in a Kubernetes installation. It allows remote, internal, or CIDR scanning over a Kubernetes cluster. It also incorporates an active option through which it tries to exploit the findings. It can be run locally or through the deployment

of a container that is already prepared with everything.

Kubesec

This tool <https://kubesec.io> allows analyzing security risk for Kubernetes resources. Among the main features we can highlight:

Helps you quantify risk for Kubernetes resources

Run against your Kubernetes applications (deployments and pods)

Can be used standalone or as kubectl plugin <https://github.com/controlplaneio/kubectl-kubesec>

Also is available as Docker container image at [docker.io/kubesec/kubesec:v2](https://hub.docker.com/r/kubesec/kubesec/tags)

<https://hub.docker.com/r/kubesec/kubesec/tags>

In the following screenshot, we can see the execution of this plugin over the Kubernetes dashboard:

```
> ~$ kubectl -n kube-system plugin scan deployment/kubernetes-dashboard
scanning deployment kubernetes-dashboard
deployment/kubernetes-dashboard kubesec.io score 3
-----
Advise
1. containers[] .securityContext .runAsNonRoot == true
Force the running image to run as a non-root user to ensure least privilege
2. containers[] .securityContext .capabilities .drop
Reducing kernel capabilities available to a container limits its attack surface
3. containers[] .securityContext .readOnlyRootFilesystem == true
An immutable root filesystem can prevent malicious binaries being added to PATH and increase attack cost
4. containers[] .securityContext .runAsUser > 10000
Run as a high-UID user to avoid conflicts with the host's user table
5. containers[] .securityContext .capabilities .drop | index("ALL")
Drop all capabilities and add only those required to reduce syscall attack surface
```

Figure 8.4: Kubesec execution over Kubernetes dashboard

In the next section, we will review different plugins that can help us, both to secure our Kubernetes cluster and to offer the best possible performance to our infrastructure.

Kubectl plugins for managing Kubernetes

There are many plugins for kubectl to interact and perform all kinds of operations against our cluster. We have seen that kubectl is the command-line tool to interact directly with Kubernetes, and it also allows you to create custom plugins, increasing your possibilities by adding ad-hoc commands to existing ones.

We are doing to review 6 plugins that offer us different security and control features to make our implementation with Kubernetes much safer. Some plugins are focused, for example, on the security of the pods, others in RABC, and we will even see one that will allow us to sniff all the network traffic generated to or from a pod.

kubectl-trace

kubectl-trace is a plugin that allows using bpftrace in a Kubernetes cluster. Thanks to bpftrace, we can create, for example, tracepoints, control points in the execution to manage its flow, or even stop it, to be able to detect problems and make an in-depth analysis of the infrastructure. These control points can be set on both nodes and pods.

In this link https://github.com/ iovisor/bpftrace/blob/master/docs/reference_guide.md, you can find the complete bpftrace manual.

You can find more information about this tool in the GitHub repository:

<https://github.com/iovvisor/kubectl-trace>

Kkubectl-debug

kubectl-debug is a plugin that complements perfectly with kubectl-trace for debugging tasks. This allows executing a container within a pod that is running. It shares the namespace of the processes (PID), network, user, and IPC of the container to be analyzed, allowing us to debug them without having to install anything beforehand. In this link, you can see a demonstration of its use <https://github.com/aylei/kubectl-debug>

debug/blob/master/docs/kube-debug.gif .

You can find more information about this tool in the GitHub repository:

<https://github.com/aylei/kubectl-debug>

Ksniff

There is another plugin called ksniff <https://github.com/eldadru/ksniff> that offers us the possibility to analyze all the network traffic of a Kubernetes pod using tcpdump and Wireshark.

In the following screenshot, we can see the execution of this plugin for getting a list of pods and verifying pod status:

```
→ ~ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
hello-minikube-7c77b68cff-qhq5b   1/1     Running   0          29m
→ ~ kubectl plugin sniff hello-minikube-7c77b68cff-qhq5b
[+] Sniffing on pod: hello-minikube-7c77b68cff-qhq5b container: namespace:
[+] Verifying pod status
NAME                  I READY   STATUS    RESTARTS   AGE
hello-minikube-7c77b68cff-qhq5b   1/1     Running   0          29m
[+] checking if tcpdump already exist
-rwxrwxr-x 1 1000 1000 2700408 Jun 22 14:08 /static-tcpdump
[+] static tcpdump is already installed on container!
[+] Starting remote sniffing!
```

Figure 8.5: Ksniff execution when verifying pod status

Ksniff uses the data collected by tcpdump associated with a pod and then send them to Wireshark to perform the analysis. This plugin is essential if you are working with microservices since it is tremendously useful for identifying errors and problems between them, as well as their dependencies.

kubectl-dig

Sometimes, getting the information from a Kubernetes cluster requires the use of several commands, which, in turn, return all kinds of information. Thanks to a plugin called kubectl-dig , it is possible to install a user-friendly UI (User Interface) to see in a simpler way all the information related to the Kubernetes cluster.

<http://github.com/sysdiglabs/kubectl-dig>

In the following screenshot, we can see the execution of this plugin for getting information from a Kubernetes cluster:

```

[15] 1:kubectl-dig*                                         "l13o1" 12:21 20-May-19
Viewing: Processes For: whole machine
Source: Live System Filter: evt.type!=switch
Select View Containers
Connections List all the containers running on this machine, and the resources that each of them uses.
Containers
Containers Errors Tips
Directories Select a container and click enter to drill down into it. At that point, you will be able to
Errors access several views that will show you the details of the selected container.
File Opens List
Files Columns
I/O by Type CPU: Amount of CPU used by the container.
K8s Controllers PROCS: Number of processes currently running inside the container.
K8s Deployments THREADS: Number of threads currently running inside the container.
K8s Namespaces VIRT: Total virtual memory for the process.
K8s Pods RES: Resident non-swapped memory for the process.
K8s ReplicaSets FILE: Total (input+output) file I/O bandwidth generated by the container, in bytes per second
K8s Services .
Marathon Apps NET: Total (input+output) network bandwidth generated by the container, in bytes per second.
Marathon Groups ENGINE: Container type.
Mesos Frameworks IMAGE: Container image name.
Mesos Tasks ID: Container ID. The format of this column depends on the containerization technology. For e
New Connections xample, Docker ID are 12 characters hexadecimal digit strings.
Page Faults NAME: Name of the container.
Processes
Processes CPU ID
Processes Errors containers
Processes FD Usage Filter
Server Ports container.name != host
Slow File I/O
Socket Queues
Spectrogram-File Action Hotkeys
Spy Syslog a: docker attach (docker attach %container.id)
Spy Users b: bash shell (docker exec -i -t %container.id /bin/bash)
System Calls f: follow logs (docker logs -f %container.id)
Threads h: image history (docker history %container.image)
Traces List i: docker inspect (docker inspect %container.id)
Traces Spectrogram k: docker kill (docker kill %container.id)
Traces Summary l: docker logs (docker logs %container.id)
s: docker stop (docker stop %container.id)
z: docker pause (docker pause %container.id)
u: docker unpause (docker unpause %container.id)
F1Help F2Views F4Filter F5Echo F6Dig F7Legend F8Actions F9Sort F12Spectro CTRL+FSearch p Pause 11/78(14.1%)

```

Figure 8.6: kube-ctl plugin execution

For the plugin execution, we only need to pass as parameter the node name, and it will obtain all detailed and formatted information about it.

Rakkess

Access control to all the elements of a Kubernetes cluster is one of the main tasks in securing it. From kubectl , it is possible to obtain this information from a resource, but it is not possible to get an overview. Rakkess plugin allows us to obtain a complete list in a matrix form of the current situation of access rights between users and all server resources.

<https://github.com/corneliusweig/rakkess>

In the following screenshot, we can see the execution of this plugin:

NAME	LIST	CREATE	UPDATE	DELETE
bindings	✓	✓	✓	✓
configmaps	✓	✓	✓	✓
controllerrevisions.apps	✓	✓	✓	✓
cronjobs.batch	✓	✓	✓	✓
daemonsets.apps	✓	✓	✓	✓
daemonsets.extensions	✓	✓	✓	✓
deployments.apps	✓	✓	✓	✓
deployments.extensions	✓	✓	✓	✓
endpoints	✓	✓	✓	✓
events	✓	✓	✓	✓
events.events.k8s.io	✓	✓	✓	✓
horizontalpodautoscalers.autoscaling	✓	✓	✓	✓
ingresses.extensions	✓	✓	✓	✓
ingresses.networking.k8s.io	✓	✓	✓	✓
jobs.batch	✓	✓	✓	✓
leases.coordination.k8s.io	✓	✓	✓	✓
limitranges	✓	✓	✓	✓
localsubjectaccessreviews.authorization.k8s.io	✓			
networkpolicies.extensions	✓	✓	✓	✓

Figure 8.7: Rakkess plugin execution

In the previous screenshot, we can see permissions for listing, creating, updating, and deleting for each resource.

Conclusion

In this chapter, we have reviewed Kubernetes security principles and some tools like Kubernetes Bench for Security project as an application that checks whether Kubernetes is implemented securely and other plugins for managing the Kubernetes cluster in a secure way.

With the objective that developers and DevOps get the best possible performance and security in the Kubernetes infrastructure, we have analyzed the state of Kubernetes security, including best practices, latest vulnerabilities discovered, and the main projects we can find in Kubernetes ecosystem for checking the security of a Kubernetes cluster.

In the next chapter, we will review Docker networking concepts, including how we can communicate and linking Docker containers.

Questions

Which is the best configuration for API authorization mode and anonymous authentication?

Which is the command google cloud provides for disabling the Kubernetes web user interface?

What is the new mechanism that Kubernetes provides to assign permissions and privileges to roles instead of specific users?

Which tool checks if Kubernetes is implemented in a secure way by executing controls documented in CIS Kubernetes Benchmark?

Which tool is a Python script developed by Aqua Security that allows analyzing the potential vulnerabilities in a Kubernetes Cluster?

CHAPTER 9

Docker Container Networking

This chapter discusses the essential components of Docker networking, including how Docker containers can be communicated and connected. Also, we will review other concepts like port mapping that Docker uses for exposing the TCP ports that provide services from the container to the host so that users accessing the host can access the services of a container.

When creating applications with containers connected to each other, we must use Docker networks to be able to communicate the containers with each other. Imagine, for example, that I want to create a blog, and for that, I need a database and an application server or something similar, at least. I could create two containers with communication between them.

Structure

Introducing container network types

Network managing in Docker

Containers communication and port mapping

Creating and managing Docker networks

Linking containers

Objectives

Knowing about container network types

Understanding network managing in Docker

Knowing about containers communication and port mapping

Knowing about creating and managing docker networks

Knowing about linking containers

Introducing container network types

Docker networking is based on the NET namespace, which allows you to generate a complete communications stack for each image running within the Docker host. In a machine where Docker is running, there are 3 network configurations. When we launch a container or set of containers that form a distributed service, we have the option to choose between these network modes.

Types of Docker networks

It is possible to change the network mode for a container using the `--net` parameter. With this parameter, you can use the default bridge (`docker0`), a different bridge, or not provide access to the network at all. Here are examples of ways to use the `--net` option:

Default network bridge: `--net = bridge` creates a new network stack for the container in the Docker bridge called `docker0`. This is the default behavior.

Without network connectivity: `--net = none` allows you to run the container without any network connection. The container is isolated from the network.

Network of another container: `--net = mycontainer` informs Docker to start the container with the capacity for using the container's network stack.

Host network: `--net = host` means that the container uses the host network stack directly from inside the container.

There are three different types of networks in Docker. To do this, you can execute the `docker network` command:

Figure 9.1: Docker network commands

With the following command, we can see networks available in the Docker host:

```
$ docker network ls
```

In the following screenshot we can see the output of the previous command:

Figure 9.2: Docker network ls

With the network mode --none the containers do not have any communication. It is used when the container is not required to have access to the external or internal network. The only IP address you have enabled is loopback or localhost.

```
$ docker run --net=none -it --rm debian
```

In the following screenshot we can see the output of the previous command:

Figure 9.3: Container network configuration with none option

In this way, your container does not have access to the network. Docker will add the container to a networking group but without a network interface. When doing a Docker inspect of a container with this network mode, we see that it is not really assigned an IP address.

```
$ docker run -it --network=none ubuntu:14.04 /bin/bash
```

```
root@18eccf3760e2:/# ifconfig  
lo Link encap:Local Loopback  
inet addr:127.0.0.1 Mask:255.0.0.0  
      UP LOOPBACK RUNNING MTU:65536 Metric:1  
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
      collisions:0 txqueuelen:1  
      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

```
$ docker inspect | grep -iaddr
```

```
"LinkLocalIPv6Address": "",  
"SecondaryIPAddresses": null,  
"SecondaryIPv6Addresses": null,  
"GlobalIPv6Address": "",  
"IPAddress": "",  
"MacAddress": "",  
"IPAddress": "",
```

"GlobalIPv6Address": "",

"MacAddress": "",

Bridge mode

The bridge mode is the default Docker network mode that will allow connectivity with the other interfaces of the host machine and between the containers. When the Docker service daemon starts, it configures a virtual Ethernet device, called docker0 . The virtual Ethernet device is mapped to appear as eth0 in the container, using the Linux namespaces.

If we start a container based on Ubuntu with this network mode, we can see how we have connectivity with both other containers of the host Docker and external internet connection.

```
$ docker run -it --network=bridge ubuntu:14.04 /bin/bash
```

In the following screenshot we can see the output of the previous command:

```
$ docker run -it --network=bridge ubuntu:14.04 /bin/bash
root@cb2d192f210c:/# ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)   TX bytes:0 (0.0 B)

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

Figure 9.4: Container network configuration with bridge option

With a simple ping command, we can check the connection with other containers in the network:

Figure 9.5: Ping connectivity inside the network container

We could also check the connection to the external network:

```
root@cb2d192f210c:/# ping google.com
PING google.com (172.217.15.78) 56(84) bytes of data.
64 bytes from iad23s63-in-f14.1e100.net (172.217.15.78): icmp_seq=1 ttl=51 time=1.38 ms
64 bytes from iad23s63-in-f14.1e100.net (172.217.15.78): icmp_seq=2 ttl=51 time=1.40 ms
64 bytes from iad23s63-in-f14.1e100.net (172.217.15.78): icmp_seq=3 ttl=51 time=1.41 ms
^C
--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
```

Figure 9.6: Ping connectivity outside the network container

The output of the bridge network could be similar to the following:

```
$ docker network inspect bridge
```

```
[{"Name": "bridge", "Id": "89d035e9545966ca300a5aed5e55fa014422a95c90519ca7d3bf55d1b1e3230e", "Created": "2019-10-28T11:53:06.650717389Z", "Scope": "local", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "default", "Options": null, "Config": [{"Subnet": "172.17.0.0/16"}]}, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {"Network": ""}}, {"ConfigOnly": false}, {"Name": "hostbridge", "Id": "hostbridge", "Created": "2019-10-28T11:53:06.650717389Z", "Scope": "host", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "host", "Options": null, "Config": null}, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {"Network": ""}}, {"ConfigOnly": false}], [{"Name": "hostbridge", "Id": "hostbridge", "Created": "2019-10-28T11:53:06.650717389Z", "Scope": "host", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "host", "Options": null, "Config": null}, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {"Network": ""}}, {"ConfigOnly": false}], [{"Name": "hostbridge", "Id": "hostbridge", "Created": "2019-10-28T11:53:06.650717389Z", "Scope": "host", "Driver": "bridge", "EnableIPv6": false, "IPAM": {"Driver": "host", "Options": null, "Config": null}, "Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {"Network": ""}}, {"ConfigOnly": false}]]
```

```

"Containers": {},
"Options": {
    "com.docker.network.bridge.default_bridge": "true",
    "com.docker.network.bridge.enable_icc": "true",
    "com.docker.network.bridge.enableipmasquerade": "true",
    "com.docker.network.bridge.hostbindingipv4": "0.0.0.0",
    "com.docker.network.bridge.name": "docker0",
    "com.docker.network.driver.mtu": "1500"
},
"Labels": {}
}
]

```

Bridge mode, through the docker0 interface, provides an internal host network in which the containers on the same host can communicate, but the IP addresses assigned to each container are not accessible from outside the host.

For instance, we could have 2 containers connected to the bridge interface of the same docker0 . A first bridge-based nginx server with IP 172.17.0.2:80 . In 172.17.0.3:80 , a second nginx container would listen, avoiding the ports conflict, as they would be different IPs.

```
$ docker run -d --name nginx-1 -p 10000:80 nginx
```

```
$ docker run -d --name nginx-2 -p 10001:80 nginx
```

In the following screenshot we can see the output of the previous commands:

Figure 9.7: Executing nginx containers

When executing Docker inspect, we can see the IP address of the container and the IP address of the gateway 172.17.0.1 .

```
$ docker inspect
```

In the following screenshot we can see the output of the previous command:

Figure 9.8: Inspecting container network configuration

In this screenshot we see the network configuration for nginx-1 container:

Figure 9.9: Inspecting nginx-1 network configuration

In this screenshot we see the network configuration for nginx-2 container:

Figure 9.10: Inspecting nginx-2 network configuration

This technique allows having several containers running on the same host without the containers having to be aware of having to listen to any dynamically assigned port.

Advantages of bridge mode:

Each container runs in its own private network namespace that is separate from the host, which increases security.

It allows containers to run on the same host without port conflicts.

Simplify the use of containers on the same host as they can all run in their ports without conflicts

Disadvantages of bridge mode:

It impacts the performance and latency of the network due to the use of NAT IP address resolution.

It requires configuring port mapping.

At this point, we have reviewed the bridge type Docker network. However, sometimes, we don't want to use the Docker network and directly use our host's network. This is possible using the --net = host argument when we deploy our container. Remember that the --net argument can be used to determine which network to use when we deploy our container.

Host mode

In this type of network, all network interfaces defined on the host will be accessible to the container; that is, the container shares the host's network namespace. To use the host network, you must execute the container with the flag --net = host:

```
$ docker run -ti --net=host busybox /bin/sh
```

The execution of the container used by the host network shows the same one occupied by the network since when executing the netstat command we see there are TCP connections in listening state on certain ports:

```
$ docker run -ti --net=host busybox /bin/sh
```

```
root@host:/# netstat -nap | head
```

In the following screenshot we can see the output of the previous command:

Figure 9.11: Container with network configuration with host option

In the following screenshot we can see active connections inside the container:

Figure 9.12: Active connections inside the container

With host mode, we can share the namespace of the host network with the container. By executing the following command, you should see the network details inside the container in the same way as they are defined in the docker host.

In the following screenshot we can see network connections inside the container:

Figure 9.13: Network connections inside the container

Since a container is just a process that runs on a host, the simplest option seems to connect it to the host's network namespace. The container will behave from the network point of view, just like any other process that runs on the host. Therefore, it will use the host's IP address and also use the host's TCP port namespace to expose the service running inside the container.

We can run a nginx container in host mode with the following command:

```
$ docker run -d --name nginx-1 --net=host nginx /bin/sh
```

This means that if our container is a NGINX web server that is listening on port 80 on the host. Imagine that later we try to run another standard web-based service on the same host. Unless otherwise indicated, our second container will probably try to connect on the same port. But since port 80 is now being used by our first container, the second container cannot be started on that host and will fail.

In the following screenshot we can see the interfaces inside the container with host mode:

Figure 9.14: Network interfaces inside the container

By executing the container with the flag `--network = host` , we can capture the traffic received by the host with the `pfcount` command.

```
$ docker run --network=host ubuntu pfcount -i eth0
```

Host mode advantages:

Easy configuration to use.

It does not perform any operation on incoming traffic (NAT, for example), so performance is not affected.

Disadvantages of host mode:

Without an additional dynamic port assignment mechanism, services can collide at the port level.

The dynamic port allocation must be managed by a container orchestration platform such as Kubernetes or Docker Swarm.

Containers share the namespace of the host network, which may have security implications. Containers that are in running state will be exposed if our Docker host is exposed to some vulnerability.

Network managing in Docker

As we have seen in the previous section, Docker offers us three different types of networks. The first bridge is where all containers would start by default. It is a network that creates a bridge between the network interface of the container that we start and a virtual network interface that is created on our computer when we install Docker.

The next one is the host mode. Host what it does is copy the host network configuration, that is, from the server or machine where Docker is in the container we are booting. If we start a container here and run the network configuration review, we will see that it is the same as the machine on which we are running it.

And finally, we have the none network, what it does is remove all the network configuration from our container. If we create a container with this configuration, we will only have the loopback address `127.0.0.1` , and we will not be able to connect any more sites.

Docker networking

Docker uses an ethernet bridge to allow the docker daemon to communicate with the machine's network device. When you build a Docker container, you can identify the ports as exposed. A container that connects to another container with an exposed port can communicate with the exposed port. To make a port accessible outside the container, you can assign a container port to a port on the host. You have to keep in mind that when you want, the container may be accessed from outside, is not enough exposing the port of a Docker container, and you need it to publish explicitly.

For example, if you expose a port, the service in the container is only accessible from inside other Docker containers. So, this feature provides inter-container communication. If you expose and publish a port, the service in the container is accessible from anywhere, even outside Docker.

One way in which Docker provides that a container is its own machine is the fact that Docker provides the container process with its own IP address. Docker does this by configuring a virtual interface and linking it to the host machine's network.

When we install Docker, we can see that it creates an interface called docker0. It has a private IP address, probably this one, if it does not collide with any other IP address that you have configured. And when you create some type of container that connects to the bridge network, what it does is receive an IP address from this range by DHCP.

All your containers will do NAT through this IP and through the IP output of the host machine or server on which you have Docker installed. This is your default network. In the same way, you can connect from here through this interface and through this IP address to the IP addresses of the containers that you have running on this network.

Among the network configurations that can be established when we execute a container, we can highlight:

--dns: A DNS server is what resolves a URL, such as <http://www.docker.io>, to the IP address of the server running the website.

--dns-search: Set up DNS search servers.

-h: Set the hostname. This will be added as an entry in the /etc/hosts file with the container IP pointing to the host.

--link: Allows a container to communicate with other containers without knowing their real IP addresses.

--expose: Expose the container port without publishing it to the host.

--publish-all: Allows publishing all ports exposed to host interfaces.

--publish: Lets you publish the port of a container for the host in the following format:

ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | containerPort

--net: This option allows you to configure the network mode for the container, as we have seen in the previous section. It can contain four values:

bridge: This creates a network stack for the container in bridge mode.

none: The container will be totally isolated and cannot communicate with any container.

container: : use the network stack of another container.

host: uses the host docker network stack.

From a container point of view, you can provide an IP-based service to other containers or applications. For doing this, you could expose the port used by the service. For example, an apache web server container should expose ports 80 and 443, as illustrated in the following Dockerfile example:

```
FROM fedora:22
```

```
MAINTAINER maintainer
```

Update the system

```
RUN yum -y update; yum clean all
```

Install httpd

```
RUN yum -y install httpd
```

```
EXPOSE 80 443 ENTRYPOINT /usr/sbin/httpd
```

For any practical implementation when dealing with ports, I suggest using port publish rather than port expose using the -p parameter to publish these ports. For example, if you want the container port 80 will be available in port 8080 of the local machine when you run the container, you need to specify:

```
$ docker run -p 8080:80
```

The previous Dockerfile uses the EXPOSE keyword to define a port that will be exposed from the container.

Expose ports: The link(Docker linking system) allows a container to access the exposed port of a container on the same machine.

Port mapping: The mapping provides a mechanism to assign an exposed port to the external ports of the host machine.

Containers communication and port mapping

When we add a container to a network, by default, all ports will be closed in the connection to the outside and all open for machines that are within the same network. For example, it would not be necessary to expose MySQL container ports, since being on the same network as the application container, they can connect through the port without a problem, but we will not be able to access the mysql port from outside the network unless we publish it.

Configure port forwarding between container and host

Port forwarding is the easiest way to expose the services that are running in containers. There are two ways to start a container and link its ports to the host ports:

-P [--publish-all]: When starting a container with this option, all the ports that were exposed using the EXPOSE statement will be published in the Dockerfile.

-p [--publish]: This option allows you to explicitly indicate to Docker which port should be linked to a port in a container. There are 3 ways to use this option:

```
$ docker run -p ip:hostport:containerport
```

```
$ docker run -p ip::container_port
```

```
$ docker run -p hostport:containerport
```

Adding an EXPOSE instruction inside a Dockerfile allows you to indicate that a specific port must be exposed from the image it builds. When a port is exposed in a running container image, it allows two things to happen:

Linked containers: Once you run the image, if you link the running container to another container, the exposed port will be available to the other container as if it were available on the same local system.

Runtime exposure: Any port identified with an EXPOSE statement when the image is built can easily be exposed from the same port number on the localhost. By using the -p option in docker run on the image, you can assign any port exposed to it or to a different specific port on the localhost. If you use the -P option in docker run, all ports exposed from the container are assigned to random ports on the host system. You can then run the docker port command in the resulting container to see how the ports are mapped.

Here is an example of a Dockerfile that launches an exposed web server on port 80 :

```
FROM fedora:latest
```

```
MAINTAINER maintainer
```

```
RUN yum install -y httpd
```

```
EXPOSE 80
```

Start the service

```
CMD ["-D", "BACKGROUND"]
```

```
ENTRYPOINT ["/usr/sbin/httpd"]
```

When you are running the container with that image, we could expose port 80 from the container to port 8080 on the local system.

```
$ docker run -p 8000:80 -d web_server
```

The -p 8000:80 parameter indicates that port 8000 on the host points to port 80 inside the container.

Exposing ports

When creating a container, you can expose the ports where it will serve. You can execute the following command for exposing ports.

```
$ docker run -itd --expose 80 --expose 443 --name
```

In order to verify the configuration of a container and view its JSON file, use the command:

```
$ docker inspect
```

In the NetworkSettings section of this JSON file you can see the port that is exposed by the container:

```
"NetworkSettings": {  
  "Bridge": "",  
  "SandboxID": "5aed72597ba63edb87532431ea432aea9ffd023899252f53acabdbc47f2af523",  
  "HairpinMode": false,  
  "LinkLocalIPv6Address": "",  
  "LinkLocalIPv6PrefixLen": 0,  
  "Ports": {  
    "443/tcp": null,  
    "80/tcp": null  
  },
```

For example, we could make the instance of a tomcat server accessible from outside the container. To do this, we need to add these flags when executing the container. First, we need to start it by specifying the mapping of ports that we want to publish with the -p (-publish) flag.

```
-p :
```

```
$ docker run -d -p 8080:8080 tomcat
```

With this simple port mapping, it is enough for the most common use cases in Docker. We will now be able to install services or microservices as Docker containers and expose their ports to enable communication.

In the previous case, we have performed a manual mapping. If we want the port to be dynamic and directly assigned by the container, Docker provides -P flag to automatically assign a port to our application:

```
$ docker run -d -P tomcat
```

In this way, we can see how the flag -p 8080:8080 it assigns a port in a manual way, and the -P flags assigns the port automatically with the mapping 32768:8080.

```
$ docker ps  
CONTAINER ID        IMAGE           COMMAND      CREATED          STATUS          PORTS          NAMES  
fc6bd72c43af        tomcat          "catalina.sh run"   2 seconds ago   Up  2 seconds   0.0.0.0:32768->8080/tcpwizardly_joliot  
6210e08cf0a2        tomcat          "catalina.sh run"   4 minutes ago   Up  4 minutes   0.0.0.0:8080->8080/tcpadmiring_banzai
```

```
$ docker port fc6bd72c43af  
8080/tcp -> 0.0.0.0:32768  
[node1] (local) root@192.168.0.38 ~  
$ docker port 6210e08cf0a2  
8080/tcp -> 0.0.0.0:8080
```

Creating and managing Docker networks

Apart from using three types of networks commented before, it is possible to create your own network configuration to use in your Docker containers. Docker allows us to create different virtual networks for our needs, either to join or segment different containers. In this way, we can separate containers for security in different networks, or join them in it for convenience or by connecting their services to each other.

Docker network commands

This is a list of the commands that can be used with Docker networking:

```
$ docker network inspect: This command lets you know the resources used by a network as well as its configuration.
```

```
$ docker network ls: It shows a list of the networks that Docker has created.
```

```
$ docker network create: It allows you to create your own network: bridge or overlay. Containers can communicate within their network but not through networks.
```

To see all the options, when creating the network, we will execute the command:

```
$ docker network create --help
```

Figure 9.15: Network create command options

In the next section, we will go into detail to create a network with the previous command.

Bridge networks

Docker's most frequently network type is a bridge network. We could create our own network for the purpose we need, for example, having a subnet in a DMZ (Demilitarized Zone) for a database server that provides service to a website that is in a public network. In the following command, we are going to create a bridge docker network.

To do this, you will have to execute the command `docker network create`.

```
$ docker network create --subnet 10.10.1.0/24 dmz
```

We could see the container assigned to the network with the `docker network inspect` command:

```
$ docker network inspect
```

This is an output example where we can see the subnet configuration and containers associated with this network.

```
{  
  "Name": "dmz",  
  "Id": "b8da7e2ab9a71f38a01e28bfbfc978e76af35fd92f300d80d4d9fae23cccd873",  
  "Created": "2019-10-28T13:27:29.657842303Z",  
  "Scope": "local",  
  "Driver": "bridge",  
  "EnableIPv6": false,  
  "IPAM": {  
    "Driver": "default",  
    "Options": {},  
    "Config": [  
      {  
        "Subnet": "10.10.1.0/24"  
      }  
    ]  
  },  
  "Internal": false,  
  "Attachable": false,  
  "Ingress": false,  
  "ConfigFrom": {  
    "Network": ""  
  },  
  "ConfigOnly": false,  
  "Containers": {  
    "b562f29b0adf2ffd6919338b22be1c5b236b3948bc780a6bd6a3bc3fe1c27906": {  
      "Name": "ubuntu",  
      "EndpointID": "201a4fe3b93f05182edc443c024c73684a52dd50f5d95a2357b7cdacca10493d",  
      "MacAddress": "02:42:0a:0a:01:02",  
      "IPv4Address": "10.10.1.2/24",  
      "IPv6Address": "",  
    }  
  },  
  "Options": {}  
}
```

```
"Labels": {}
```

```
}
```

```
]
```

Connect container to a network

In order to connect a container to a network we must specify to which network we want to connect it with the --network option followed by the name of the network to which we want to add it. For example, to run a MySQL container and add it to the network that we just created, the command would be the following: docker container run -d --name blog_mysql --network blog mysql .

To add a container that already exists to a network, we must use the following command:

```
$ docker network connect
```

With the command docker network connect you can connect a container to an available network.

After connecting a container to a network, we can see its configuration by inspecting the configuration of the container with the command:

```
$ docker network connect
```

If we inspect the container, we can see that we have two network interfaces, one for the bridge default and the second one for the network created called dmz .

```
"Networks": {
```

```
    "bridge": {
```

```
        "IPAMConfig": null,
```

```
        "Links": null,
```

```
        "Aliases": null,
```

```
        "NetworkID": "c3b1c0061755c81c023825a9c232b7a6fcfb546fedc7143e63f21ac4e443b545",
```

```
        "EndpointID": "3ded099d19af8b01ac74a549ba9847dd32cbc0e16fd21f27c4b928a39739d25",
```

```
        "Gateway": "172.17.0.1",
```

```
        "IPAddress": "172.17.0.2",
```

```
        "IPPrefixLen": 16,
```

```
        "IPv6Gateway": "",
```

```
        "GlobalIPv6Address": "",
```

```
        "GlobalIPv6PrefixLen": 0,
```

```
        "MacAddress": "02:42:ac:11:00:02",
```

```
        "DriverOpts": null
```

```
    },
```

```
    "dmz": {
```

```
        "IPAMConfig": {},
```

```
        "Links": null,
```

```
        "Aliases": [
```

```
            "b562f29b0adf"
```

```
        ],
```

```

"NetworkID": "b8da7e2ab9a71f38a01e28bfbfc978e76af35fd92f300d80d4d9fae23cccd873",
"EndpointID": "201a4fe3b93f05182edc443c024c73684a52dd50f5d95a2357b7cdacca10493d",
"Gateway": "10.10.1.1",
"IPAddress": "10.10.1.2",
"IPPrefixLen": 24,
"IPv6Gateway": "",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"MacAddress": "02:42:0a:0a:01:02",
"DriverOpts": {}

}
}

```

In this other example we are going to create a network called my-network:

```

$ docker network create my-network
79896668b483d38ca83642f3197afce9188116d079b70203c065c053dfe8980
[node1] (local) root@192.168.0.28 ~
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
80302a34d531    bridge    bridge      local
21276d934921    host      host       local
79896668b483    my-network    bridge      local
03bf600bbfec    none      null       local

```

Figure 9.16: Creating a networking

We could start a container based on Elasticsearch image with the name elasticSearch and connect it to the new network we have created:

```

$ docker run -d --net=my-network --name elasticSearch elasticsearch:2.2
$ docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED
STATUS      PORTS      NAMES
8c7c805209c2      elasticsearch:2.2      "/docker-entrypoint..."      About a
minute ago      Up About a minute      9200/tcp, 9300/tcpelasticsearch

```

We could also start a container based on Ubuntu with the name ubuntu and connect it to the new network. In this case we will start a container based on an ubuntu image:

```
$ docker run -it --net=my-network --name ubuntu ubuntu:14.04 bash
```

The next step is to verify the visibility between both containers. To do this, we first connect to the ubuntu container using the docker exec command:

```
$ docker exec -it ubuntu bash
```

We could ping and request the elasticSearch container from the ubuntu container with the commands:

```
$ ping elasticSearch
```

```
$ curl http://elasticSearch:9200
```

In the following screenshot we can see the output of the previous commands:

Figure 9.17: Checking connectivity with elasticSearch container

We could also check the connection to the Ubuntu container from the elasticSearch container:

```
$ docker exec -it elasticSearch bash
```

```
root@8c7c805209c2:/usr/share/elasticsearch# ping ubuntu
```

```
PING ubuntu (172.19.0.3): 56 data bytes
```

```
64 bytes from 172.19.0.3: icmp_seq=0 ttl=64 time=0.089 ms
```

```
64 bytes from 172.19.0.3: icmp_seq=1 ttl=64 time=0.080 ms
```

```
64 bytes from 172.19.0.3: icmp_seq=2 ttl=64 time=0.072 ms
```

```
^C--- ubuntu ping statistics ---
```

```
3 packets transmitted, 3 packets received, 0% packet loss
```

```
round-trip min/avg/max/stddev = 0.072/0.080/0.089/0.000 ms
```

In this way, we have reviewed how to create new networks in Docker and connect new containers that can communicate with each other.

Linking containers

When a container node is created, it is necessary that these containers can be connected to each other by IP address or by hostname. But if a container is turned off when started again, new parameters are generated, such as the ID and the IP address it uses.

In order to overcome this problem, there is the functionality of linking one or more containers, that will allow each time one of the linked containers is turned off and on, the IP assigned by the docker engine does not matter since it is assigned will connect by container name. All the necessary magic is done by the Docker engine, such as routing rules, DNS rules, etc.

Linking containers within the same host with --link

Links between containers are based on their names, so it is often useful to use meaningful names to name our containers. Network connections between containers is another system to establish connections between containers and use it for data exchange.

What we are going to see here is how to establish links between containers using this linking system. In this linking system, one container acts as a data source and the other receiver container. This is similar to the concept of pipe in Unix system.

The link allows a container to communicate with another container without knowing its IP address. This is achieved by inserting the IP address of the first container in the /etc/hosts file of the second container. In order to link containers, the variable –the link must be used when creating a container with the command:

```
$ docker run --link --name -h
```

When using the --link flag, Docker adds an entry to the /etc/hosts file of the container, with the hostname, the IP address of the container, and the identifier .

We could create an elasticSearch container with the command:

```
$ docker run -d --name elasticSearch elasticsearch:2.2
```

Next, we create aUbuntu container using the link tag to connect this container with the elasticSearch we created previously.

```
$ docker run -it --name ubuntu --link elasticSearch:elasticSearch ubuntu:14.04 bash
```

After executing the previous command, we can check how we have connectivity between the ubuntu container and elasticSearch . Not so between elasticSearch and Ubuntu, since the links are established in the sense of a source container and a destination container in a unidirectional way.

What is happening here is that Docker creates a secure tunnel between both containers, preventing the ports of both are exposed outside the container and are therefore accessible from our host or any other container.

In this screenshot we can see the content of the etc/hosts file inside the Ubuntu container:

```
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      elasticSearch 3f7c7e796cd5
172.17.0.3      a05699ed73ab
```

Figure 9.18: etc/hosts file inside the Ubuntu container

When we create a link, Docker is responsible for updating the /etc/hosts file in order to access the container on which we establish the link. If we go to the contents of the /etc/hosts file of the Ubuntu container, we can see the reference to the elasticSearch container.

Environment variables

In addition to modifying the /etc/hosts file, Docker creates in the container where we establish the link (Ubuntu) some environment variables with the information of the other container (elasticSearch). Among the information that Docker makes available using environment variables is the IP of the other container, in this case, the elasticSearch .

For example, if we look at all the environment variables of the Ubuntu container, we will see all related to elasticSearch :

```
root@a05699ed73ab:/# set | grep -ielasticSearch
```

In the following screenshot we can see the output of the previous command:

```

root@a05699ed73ab:/# set | grep -i elasticSearch
ELASTICSEARCH_ENV_CA_CERTIFICATES_JAVA_VERSION=20140324
ELASTICSEARCH_ENV_ES_VERSION=2.2.2
ELASTICSEARCH_ENV_GOSU_VERSION=1.7
ELASTICSEARCH_ENV_JAVA_DEBIAN_VERSION=8u111-b14-2~bpo8+1
ELASTICSEARCH_ENV_JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre
ELASTICSEARCH_ENV_JAVA_VERSION=8u111
ELASTICSEARCH_ENV_LANG=C.UTF-8
ELASTICSEARCH_NAME=/ubuntu/elasticSearch
ELASTICSEARCH_PORT=tcp://172.17.0.2:9200
ELASTICSEARCH_PORT_9200_TCP=tcp://172.17.0.2:9200
ELASTICSEARCH_PORT_9200_TCP_ADDR=172.17.0.2
ELASTICSEARCH_PORT_9200_TCP_PORT=9200
ELASTICSEARCH_PORT_9200_TCP_PROTO=tcp
ELASTICSEARCH_PORT_9300_TCP=tcp://172.17.0.2:9300
ELASTICSEARCH_PORT_9300_TCP_ADDR=172.17.0.2
ELASTICSEARCH_PORT_9300_TCP_PORT=9300
ELASTICSEARCH_PORT_9300_TCP_PROTO=tcp

```

Figure 9.19: ElasticSearch variables file inside the Ubuntu container

As we see, all the information in the elasticSearch container is available in the ubuntu container, so using the environment variables, we can access and discover the services of another container. This is sometimes useful, and it can be a security problem when we want to hide some of the services from one container to another.

In this other example, we are going to communicate MySQL with wordpress. The first command executes the mysql container. In this case, we provide an environment variable so that the container can initialize the database.

```

$ docker run --name wp-mysql -e MYSQL_ROOT_PASSWORD=yoursecretpassword -d mysql
$ docker run --name wordpress --link wp-mysql:mysql -p 10003:80 -d wordpress

```

In the second command, run a container with the Wordpress image. Using the --link tag, we link the mysql container to the wordpress container (--link wp-mysql: mysql). It also uses a local port assignment (-p 10003:80). If you now navigate to the <http://localhost:10003> URL address, you will see the Wordpress screen.

In the following screenshot we can see the containers in execution:

Figure 9.20: Containers Wordpress and MySQL in execution

In this other example we create a container based on the image of Redis and link it with a container based on Debian Linux distribution:

First, we create a redis container with the name myredis :

```
$ docker run -d --name myredisredis
```

c9148dee046a6fefac48806cd8ec0ce85492b71f25e97aae9a1a75027b1c8423

Next, we link redis container with debian container:

```
$ docker run --link myredis:redisdebian env
```

In the following screenshot we can see the output of the previous command:

Figure 9.21: Redis variables inside the Debian container

We can see that Docker has configured environment variables with the prefix REDIS_PORT inside the Debian container, which contains information on how to connect to the Redis container.

Docker has also imported environment variables from the linked container, which has the prefix REDIS_ENV . While this functionality can be very useful, it is important to keep in mind that if you use environment variables to store secrets such as API tokens or database passwords, this data can be exposed in other containers.

Conclusion

In this chapter, we have reviewed how networks are configured in Docker containers that should not be disconnected from other systems, whether physical, virtual, or in containers. The reader has learned the main types of Docker networks and how connecting containers to each other with the creation of your own Docker network. Thanks to the use of the Docker networks, we have the ability to create more complex applications, with the possibility that each container offers a service that works autonomously, and the containers can communicate with each other. That is why Docker provides commands for managing Docker networks.

In the next chapter, we will review open-source tools available for Docker container monitoring such as cAdvisor, dive, and sysdigFalco.

Questions

Which is the default Docker network mode that will allow connectivity with the other interfaces of the host machine and between the containers?

In which type of network, all network interfaces defined on the host will be accessible to the container, and the container will share the host's network namespace?

Which flag allows you to explicitly indicate to Docker which port should be linked to a port in a container?

Which Dockerfile instruction allows you to indicate that a specific port must be exposed from the image it builds?

Which is the command you can use for connecting a container to an available network?

CHAPTER 10

This chapter introduces some of the open-source tools available for Docker container monitoring such as cAdvisor, dive, and Sysdigfalco.

When you run Docker in production, one of the important things to consider is how to measure the performance of the containers. It is important to define a comprehensive strategy to monitor your Docker infrastructure with a native collection source for events, statistics, configurations and records, and provide views on the performance of the CPU, memory, and network containers.

Structure

Container statistics, metrics and events

Performance monitoring with cAdvisor

Performance monitoring with Dive

Container monitoring with Sysdigfalco

Objectives

Know about obtaining statistics, metrics, and events from Docker containers

Know about cAdvisor as a performance monitoring tool

Know about Dive as a performance monitoring tool

Know about Sysdigfalco as Container monitoring tool

Container statistics, metrics and events

There are several ways to control the execution of Docker containers. It is possible to visualize the logs, observe the events and statistics of the container at the level of memory usage and CPU. Let's see what Docker offers when we want to visualize the logs that are recorded when we execute a container.

Log management

Most applications send logs to the standard output. If the container is running in the foreground, you can see the log directly in the console. However, when running a container in background mode, only the container identifier (ID) will be displayed on the console.

Log management is one of the most important tasks in the world of security, as it allows monitoring what is happening inside containers. Different containers will run simultaneously in the same Docker host, and each of them can generate their own logs; therefore, the centralized management of the logs is necessary.

There are several commands for monitoring the logs:

```
$ docker logs
```

```
$ docker service logs
```

In this case, the Docker engine collects in a log file all the standard output of a running container. We can visualize the execution log of a container with the following command:

```
$ docker logs -f
```

In the following screenshot we can see the logs command options:

Figure 10.1: Docker logs command options

The way this works is that logs sent to the standard output or error output in the container are captured by the Docker daemon process and transmitted to a configurable backend, which is by default a JSON file for each container. In this example, we can see the log output of a nginx container.

```
$ docker logs nginx

nginx stderr | 2019/11/09 00:34:56 [notice] 12#0: using the "epoll" ...
nginx stderr | 2019/11/09 00:34:56 [notice] 12#0: nginx/1.0.15
nginx stderr | 2019/11/09 00:34:56 [notice] 12#0: built by gcc 4.4.7 ...
nginx stderr | 2019/11/09 00:34:56 [notice] 12#0: OS: Linux 3.8.0-35-generic
```

The files that support this log are located on the Docker host by default in the path /var/lib/docker/containers/ where the is replaced by the container identifier. The format of these files is similar to one where each line is represented by a JSON object.

In the following screenshot, we can see the path where logs are located for each container:

Figure 10.2: Path where logs are located for each container

By default, logs are stored in a JSON file located in the path /var/lib/docker . This behavior can be changed since Docker uses the concept of registry drivers. By using different controllers, it is possible to choose another type of storage for logging. The default driver is the JSON file, which accepts the following configuration:

```
--log-opt max-size = [0-9+] [k|m|g]
--log-opt max-file = [0-9+]
```

The previous command options can be used where the standard output (STDOUT) is correctly configured. In some instances, the use of the above commands will not be adequate because the data is not available in an appropriate format. In these cases, the following steps need to be followed:

In the case that a process is being used within a running container to handle the logs, it would not be advisable to use the docker logs command since the required information will not be displayed.

In the case that a non-interactive process such as a web server, is being executed within the container, it could have a service that is sending logs to a file; therefore, the conventional outputs will not be enabled. One solution is making a redirection of conventional logs.

Below are the options to redirect and format the logs so that they can be used in the best possible way. These are the different drivers that Docker supports:

For using any of these driver controllers, you must use the --log-driver option when executing the docker run command. For example, to store log entries in the syslog of an nginx-based container, we could execute:

```
$ docker run --log-driver=syslog nginx
```

Observing logs is the most convenient way to monitor our application on the Docker host. We could also see the properties of the running containers, such as the mapped network port or the volume being mapped. It is more efficient to use the docker inspect command to display these resources, which provides all of this information in the form of metadata.

Stats in containers

The stats command allows you to obtain statistics for one or more containers in execution in real-time. This command allows you to see the use of CPU, memory, I/O operations at the network level. The syntax for the command is as follows:

```
$ docker stats [OPTIONS] [CONTAINER...]
```

In the following screenshot we can see the stats command options:

Figure 10.3: Docker stats command options

The previous command works through the Docker daemon process that obtains cgroups resource information and serves it through the APIs. By default, if no containers are specified, the command will display statistics for all running containers.

```
$ docker ps
```

```
$ docker stats
```

For example, if we have two containers running MySQL and Ubuntu, this command will show us the statistics of both.

```
$ docker stats
```

In the following screenshot we can see the output of the previous command:

Figure 10.4: Docker stats in MySQL and Ubuntu containers

With docker stats , we also have real-time statistics of all the containers running. The docker stats command accepts the following options:

--no-stream: This option disables real-time statistics and will only show the first result.

-a (--all): This option shows the statistics of all containers.

Statistics can be used to see the behavior of containers during execution. The information can be useful to verify if we need some restrictions on the resources that will be applied to the containers.

Viewing records, container metadata, and runtime statistics offer many possibilities when monitoring running containers. We can also see what happens in the Docker host when it receives a command, and it will issue an event that we can observe.

```
$ docker stats
```

```
CONTAINER CPU% MEM USO / LIMIT MEM% NET I / O
```

```
0.00% 7.227 MiB / 987.9 MiB 0.73% 936 B / 468 B
```

Basically, it provides information about the amount of CPU it consumes a container, the amount of memory it has in use, and the limit of what it can use. You can also see the percentage of memory used to make it easier for the user to check how much free memory the container has available.

The endpoint provides the statistics in a more detailed way in JSON format.

For example, we can start aUbuntu container and view its statistics with the following command:

```
$ docker run -d ubuntu:latest sleep 1000
```

```
91c86ec7b33f37da9917d2f67177ebfaa3a95a78796e33139e1b7561dc4f244a
```

Now that the container is running, we can access the endpoint /stats on port 2375. To access this information, the container identifier obtained after executing the preceding command needs to be passed in the URL.

```
$ curl -s http://localhost:2375/v1.40/containers//stats
```

In the following screenshot we can see the output of the previous command:

Figure 10.5: Docker stats through the endpoint

In this endpoint, you can get detailed memory usage information, as well as information about CPU usage. Keep in mind that the endpoint is executed by container, so it is not possible to obtain the statistics of all the containers of a single call using this endpoint.

If you have a specific metric that you want to monitor, you can write your own solution using runc or making calls to the kernel directly. You will have to use a language that allows you to make low-level kernel calls, such as Go or C.

The Runtime Metrics project <https://docs.docker.com/config/containers/runmetrics/> shows how to do it and delves into the various kernel metrics available. Once you have exposed the values you need, you can search for tools such as statsd to add metrics, InfluxDB and OpenTSDB for metric storage, and Grafana to display the results.

Obtain metrics using docker inspect

Another way to obtain metrics is through the docker inspect command, where the ps -q option allows you to get the identifiers of all the containers in execution.

```
$ docker stats --no-stream $(docker ps -q) | sed -e "$!(docker ps --format \"{{.ID}} {{.Names}}\")" | sed -e "s/(.) (.)/s\1\2\t\2/g;" "
```

In the following screenshot we can see the output of the previous command:

Figure 10.6: Docker stats with docker inspect

In this section, we have reviewed the docker stats command to forget the main statistics inside a Docker container. Next section, we will focus on other commands for getting the events generated inside a container.

The Docker daemon process internally generates a flow of events around the container's life cycle. With the docker events command, it is possible to see what life cycle events are happening in real-time inside the container.

Events in Docker containers

The sequence of events is useful for monitoring scenarios and performing additional actions, such as receiving an alert when a task ends. In the case of running many containers in production, it will be useful if we can see container events in real-time for monitoring and debugging purposes.

In the following screenshot, we can see the events command options:

Figure 10.7: Docker stats with docker inspect

To observe the events that arrive at the Docker engine in real-time, we use the docker events command. This command can be useful if we want to know what happened during the runtime of the container. Containers report a large list of events.

The list includes the following commands: attach, commit, copy, create, destroy, detach, die, execcreate, execdetach, exec_start .

The event command contains the -f/--filter parameter, which allows you to filter the result if you are looking for something specific. If no filter is provided, all events will be reported.

Calling the events command will show what activities occur in Docker, processes that are running in real-time for tracking almost all actions and system calls captured like events.

The list of possible filters includes:

container (container=)

event (event=)

image (image=)

plugin (experimental) (plugin=)

label (label= or label==)

type (type=)

volume (volume=)

network (network=)

daemon (daemon=)

You can use the --since or --until option with Docker events to filter the results from a timestamp or timestamp:

--since = "Date" Show all events created from a date

--until = "Date" Show all events created up to a date

For example, the following command shows events from starting the year 2020:

```
$ docker events --since '2020-01-01'
```

We can also obtain the events of a specific container from its identifier ID:

```
$ docker events --filter container=
```

In this screenshot we can see the output of applying date and filter conditions:

Figure 10.8: Docker stats with docker inspect

In the previous output, we can see the events that have registered for creating, attaching and starting a container.

In the official documentation, we have available in detail the possibilities offered by the events command:
<https://docs.docker.com/engine/reference/commandline/events> .

Other Docker container monitoring tools

Within the docker ecosystem, we can find other tools such as ctop and LazyDocker.

ctop <https://github.com/bcicen/ctop> is a tool developed in Golang that provides an overview of real-time metrics for multiple containers in a graphical way. Also, the source code is available in the GitHub repository: <https://github.com/bcicen/ctop> .

You can install it downloading the latest version with the command:

```
$ wget https://github.com/bcicen/ctop/releases/download/v0.7.2/ctop-0.7.2-linux-amd64-O/usr/local/bin/ctop
```

We give execution permissions with the command:

```
$ chmod +x /usr/local/bin/ctop
```

Also is available as Docker image and you can execute it with the command:

```
$ docker run --rm -ti \
```

```
--name=ctop \
```

```
--volume /var/run/docker.sock:/var/run/docker.sock:ro \
```

```
quay.io/vektorlab/ctop:latest
```

In the following screenshot we can see the output of the previous command:

ctop - 13:51:59 UTC 3 containers							
NAME	CID	CPU	MEM	NET RX/TX	IO R/W	PIDS	
● ctop	060a55c5244c	1%	9M / 31.4G	0B / 0B	0B / 0B	13	
● laughing_greid...	c92da791b0af	0%	5M / 31.4G	0B / 0B	32K / 0B	2	
● sad_bouman	db2de5b34fc2	-	-	-	-	-	

Figure 10.9: Execution of ctop command

In the previous screenshot, we can see the containers in execution. Also, we have other options for getting a single view, stop and restart a specific container:

3 containers						
	CPU	MEM	NET RX/TX	IO R/W	PIDS	
[o] single view						
[l] log view	0a55c5244c	1%	9M / 31.4G	0B / 0B	0B / 0B	13
[s] stop	2da791b0af	0%	5M / 31.4G	0B / 0B	32K / 0B	2
[p] pause		-	-	-	-	-
[r] restart	2de5b34fc2					
[e] exec shell						
[c] cancel						

Figure 10.10: Get visualization options

If we select a single view, we can see the container details, usage of CPU, and memory:

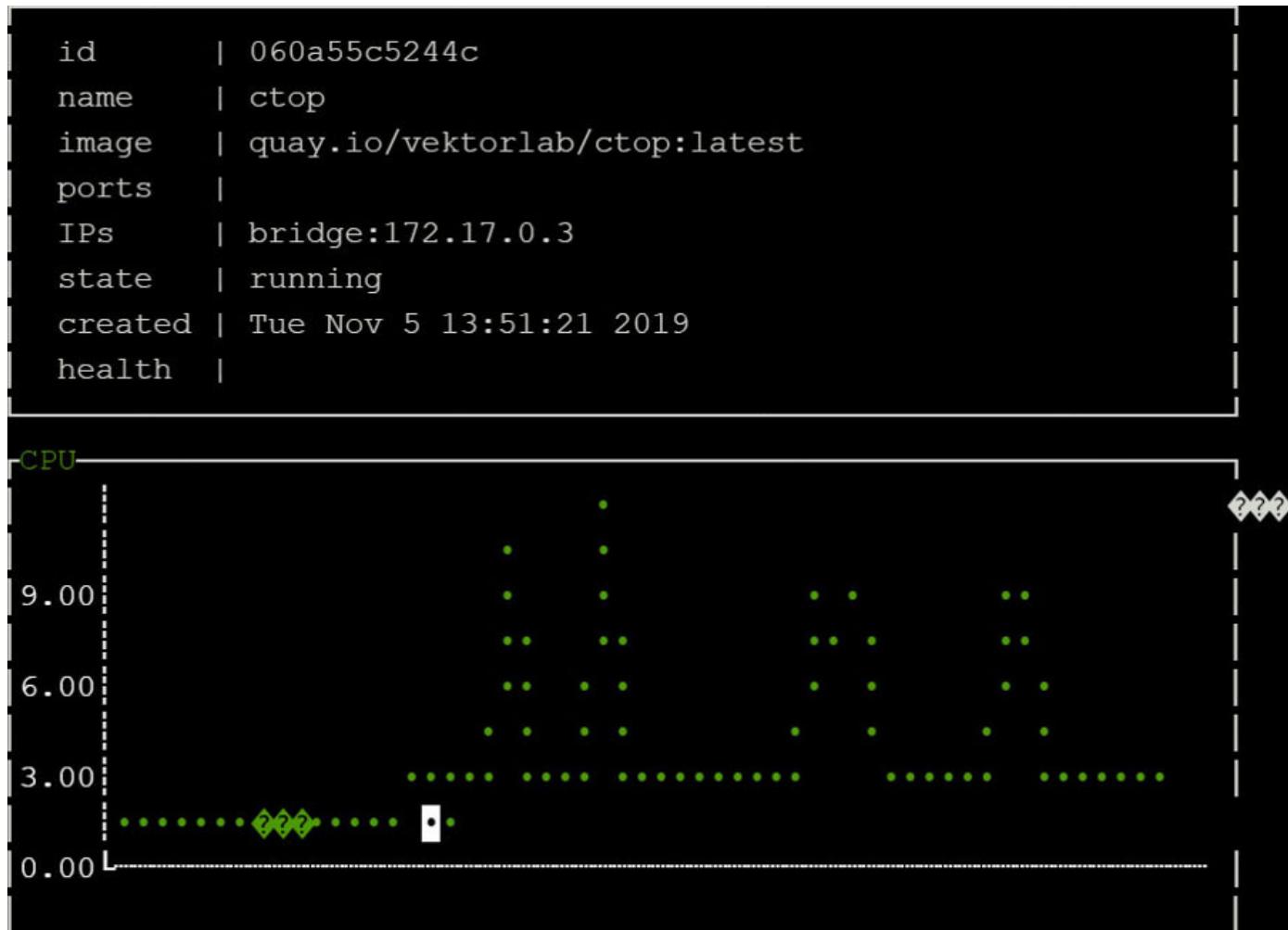


Figure 10.11: Show container details and usage of CPU

LazyDocker is a terminal user interface for both docker and docker-compose , written in Go with the gocui library. You can find source code and installation instructions in the GitHub repository: <https://github.com/jesseduffield/lazydocker> .

You can simplify the installation and execution of this tool using the docker-compose .yml file you can find in the GitHub repository:

<https://github.com/jesseduffield/lazydocker/blob/master/docker-compose.yml>

```
version: '3'
```

```
services:
```

```
lazydocker:
```

build:

context: https://github.com/jesseduffield/lazydocker.git

args:

BASE/MAGEBUILDER: golang

GOARCH: amd64

GOARM:

image: lazyteam/lazydocker

container_name: lazydocker

stdin_open: true

tty: true

volumes:

- /var/run/docker.sock:/var/run/docker.sock
- ./config:/config/jesseduffield/lazydocker

For executing the previous file, we can use docker-compose up -d command.

In the following screenshot we can see the output of executing the previous docker-compose file:

The screenshot shows the LazyDocker interface. On the left, there's a tree view with sections: Project (containing a single slash '/'), Containers (listing 'running', 'sweet_torvalds', and 'lazyte'), Images (listing various Docker images with their details like ID, Tag, Size, and Command), and Volumes (empty). On the right, there's a detailed view for the 'lazyteam/lazydocker' container. It shows the container's configuration: Name: lazyteam/lazydocker, ID: sha256:02b45fa1be32a3ae8f7b3bdb79930cd6e301f606cf0f2f5ea17, Tags: lazyteam/lazydocker:latest, Size: 69.94MB, and Created: Tue, 05 Nov 2019 08:18:57 UTC. Below this, it lists the images used by the container, including their IDs, tags, sizes, and commands.

ID	TAG	SIZE	COMMAND
02b45fa1be	lazyteam/lazydocker:latest	10.19MiB	COPY file:3836b2a7104
	<missing>	56.52MiB	COPY file:6048005b542
arm32v7/golang	1.12.6-alpine3	0B	ENTRYPOINT ["/bin/laz
arm32v6/golang	1.12.6-alpine3	0B	LABEL org.opencontain
arm64v8/golang	1.12.6-alpine3	0B	ARG VERSION
	<missing>	0B	ARG VCS_REF
	<missing>	0B	ARG BUILD_DATE

Figure 10.12: Show container details with LazyDocker

When executing LazyDocker, you can see information related to containers that are executing and the layers generated for each image.

Performance monitoring with cAdvisor

In the Docker ecosystem, we can find some tools that allow visualizing in a graphic way the use of CPU and memory of the containers in execution in our host Docker. Among which we can highlight cAdvisor, Prometheus, and Dive.

cAdvisor as a monitoring tool

cAdvisor is one of the best useful tools that enable container-oriented performance monitoring. Among other things, it allows monitoring:

Resource isolation parameters

Historical use of resources

Network statistics

The tool is also available as a public image in the Docker Hub: <https://hub.docker.com/r/google/cadvisor>

Also, the source code is available in the GitHub repository:

<https://github.com/google/cadvisor>

It is possible to run cAdvisor as if it were an application. For this, we need to download the source code and run the application in standalone mode following the documentation:

<https://github.com/google/cadvisor/blob/master/docs/running.md#standalone>

The easiest way to run cAdvisor is to do it through a container with the execution of the following command:

```
sudo docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:ro \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker:/var/lib/docker:ro \
--volume=/dev/disk/:/dev/disk:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

In the following screenshot we can see the output of the previous command:

Figure 10.13: cAdvisor execution

After having started the container with the previous command, we can access from the browser the URL <http://localhost:8080> without requesting a user or password. If we need some kind of authentication for this service, we can use the project: <https://github.com/tim545/docker-cadvisor>

basicauth .

For using this project, first, we need to clone it:

```
$ git clone https://github.com/tim545/docker-cadvisor-basicauth
```

Now, we can build the Docker image, establishing username and password:

```
$ docker build --build-arg USERNAME=admin --build-arg PASSWORD=Password1 -t tim545/cadvisor-basicauth.
```

```
$ docker run \
```

```
--volume=/:/rootfs:ro \
```

```
--volume=/var/run:/var/run:rw \
```

```
--volume=/sys:/sys:ro \
```

```
--volume=/var/lib/docker:/var/lib/docker:ro \
```

```
--publish=8080:8080 \
```

```
--detach=true \
```

```
--name=cadvisor-basicauth \
```

```
--restart=always \
```

```
tim545/cadvisor-basicauth:latest
```

And we can already access the container, through the IP of our docker server on the port 8080 . In this case we are using basic authentication and we need enter user and password.

This application allows visualizing the use of CPU and memory graphically. In the Docker Containers section, you can see the URLs of the containers that are running on the Docker host. If you click on any of them, you will see the resource usage information for the corresponding container.

In the following screenshot we can see the cAdvisor main screen:

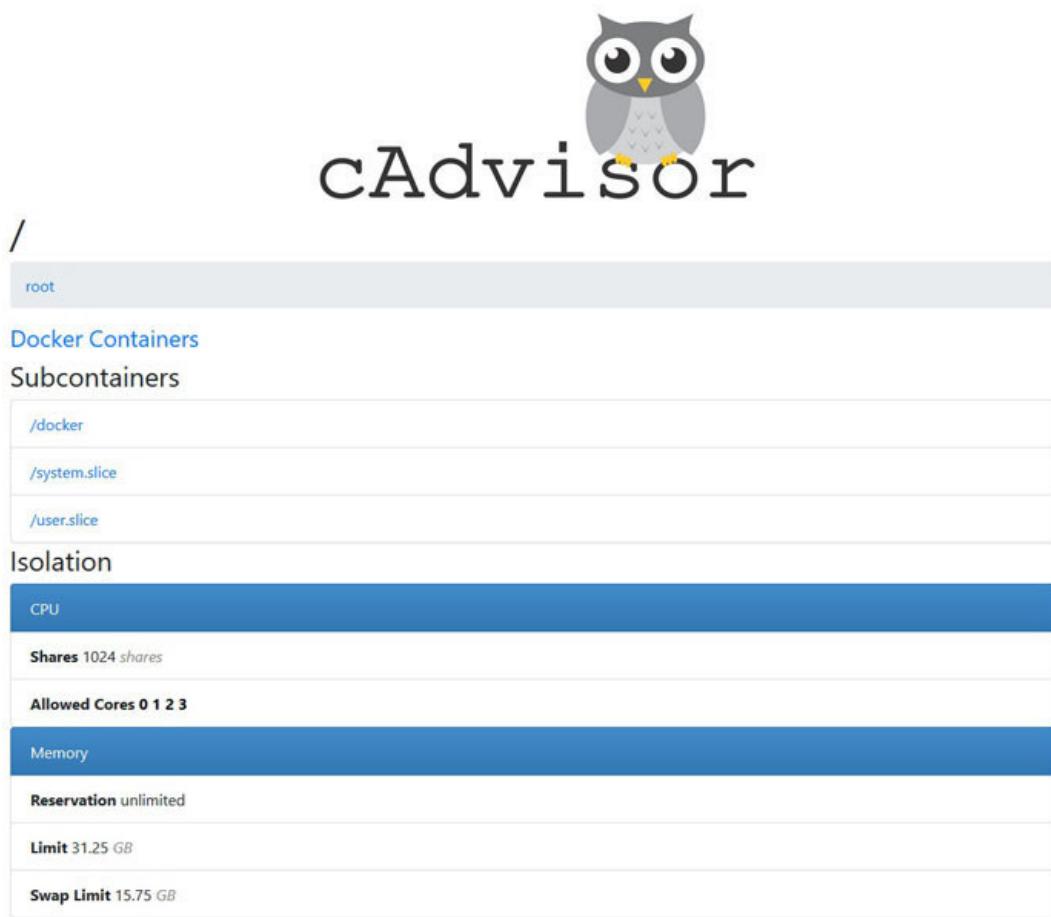


Figure 10.14: Showing information about containers in execution

With the docker run command, we have mounted some volumes from host machines in read-only mode. cAdvisor will read the relevant information from those such as cgroup details for containers and display them graphically.

cAdvisor provides an endpoint in the form of a REST API, where you can query all the information provided by the containers:

```
$ curl http://localhost:8080/api/v1.3/containers
```

In the following screenshot we can see processes that are running and the use of the CPU and memory:

Usage

Overview



Processes

User	PID	PPID	Start Time	CPU %	MEM %	RSS	Virtual Size	Status	Running Time	Command	Container
root	21.799	21.781	15:52	3.30	0.20	79.45 MiB	1.19 GiB	Rsl	00:00:12	cadvisor	/docker /89db4ec304d30dde00c6a7
root	5.166	1	Feb25	0.90	0.30	115.30 MiB	1.59 GiB	Ssl	00:30:17	dockerd	/system.slice /docker.service
root	5.236	1	Feb25	0.30	0.10	42.13 MiB	1.12 GiB	Ssl	00:10:15	containererd	/system.slice /containererd.servi
root	9.787	9.701	Feb25	0.30	0.10	38.20 MiB	732.30 MiB	Ssl	00:09:39	telegraf	/docker /dfb008f7351e6447bcafd5
root	23.689	23.686	10:57	0.20	0.90	293.71 MiB	8.46 GiB	S1	00:00:40	java	/docker /ba32db3e25d30b389564a
root	23.833	23.830	10:57	0.20	0.90	291.14 MiB	8.46 GiB	S1	00:00:41	java	/docker /cb534d308dc18d2a6462f5
root	29.421	29.420	10:12	0.20	0.80	278.55 MiB	10.33 GiB	S1	00:00:43	java	/docker /c8dbe3c1d899c5658310bb
root	29.579	29.578	10:12	0.20	0.80	273.49 MiB	10.33 GiB	S1	00:00:43	java	/aeff16f0b2253a1a729b688
root	1.566	1.536	15:14	0.10	0.00	1.97 MiB	111.04 MiB	Ss	00:00:04	bash	/user.slice /docker /757db37c3fe051d0779286
root	25.741	25.697	10:06	0.10	0.80	286.04 MiB	10.33 GiB	S1	00:00:38	java	/aeff16f0b2253a1a729b688
root	25.891	25.842	10:06	0.10	0.90	292.18 MiB	10.33 GiB	S1	00:00:38	java	/4f8a0664245c4a37ef4255
root	30.840	30.790	10:13	0.10	1.00	321.45 MiB	8.47 GiB	S1	00:00:41	java	/5d65cec329c52416d15651
root	30.979	30.933	10:13	0.10	1.00	321.80 MiB	8.47 GiB	S1	00:00:41	java	/1c74596fcbad96fa966af3
root	1	0	Feb25	0.00	0.00	4.03 MiB	186.63 MiB	Ss	00:00:38	systemd	/
root	2	0	Feb25	0.00	0.00	0.00 B	0.00 B	S	00:00:00	kthreadd	/
root	3	2	Feb25	0.00	0.00	0.00 B	0.00 B	S	00:00:01	ksoftirqd/0	/
root	5	2	Feb25	0.00	0.00	0.00 B	0.00 B	S<	00:00:00	kworker/0:0H	/
root	7	2	Feb25	0.00	0.00	0.00 B	0.00 B	S	00:00:01	migration/0	/
root	8	2	Feb25	0.00	0.00	0.00 B	0.00 B	S	00:00:00	rcu_bh	/
root	9	2	Feb25	0.00	0.00	0.00 B	0.00 B	S	00:01:43	rcu_sched	/
root	10	2	Feb25	0.00	0.00	0.00 B	0.00 B	S<	00:00:00	lru-add-drain	/

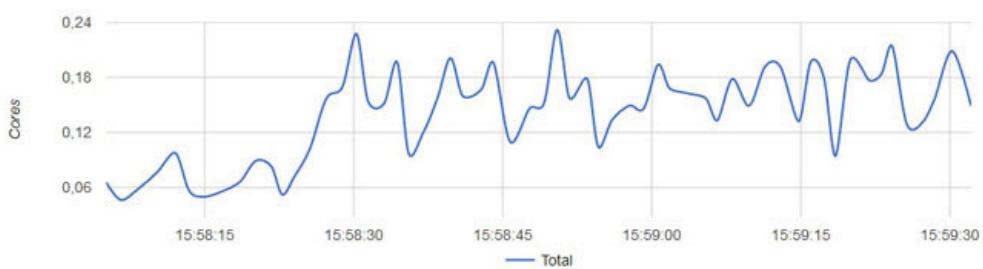
Figure 10.15: Showing information about processes

If we click on subcontainers /docker , it will show us a list of all the containers, and we can click on each one to see their statistics individually.

Also, we can get detail information related to the use of CPU per core:

CPU

Total Usage



Usage per Core

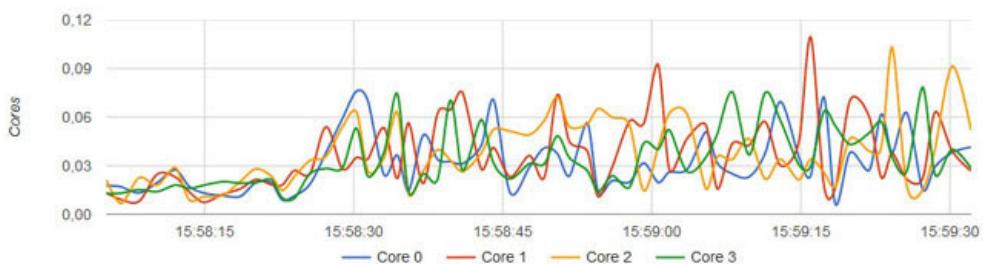
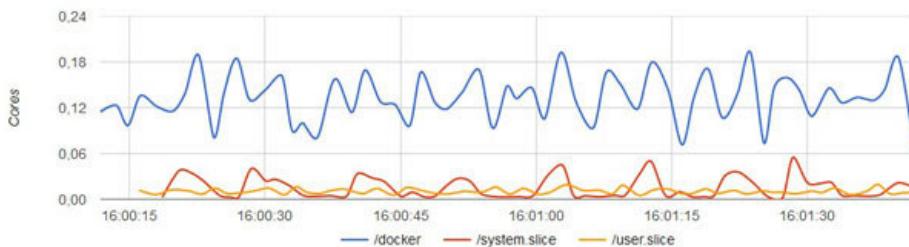


Figure 10.16: Usage of CPU per core

Also, we can get detail information related to the usage of CPU and memory for each container:

Subcontainers

Top CPU Usage: 10



Top Memory Usage: 10

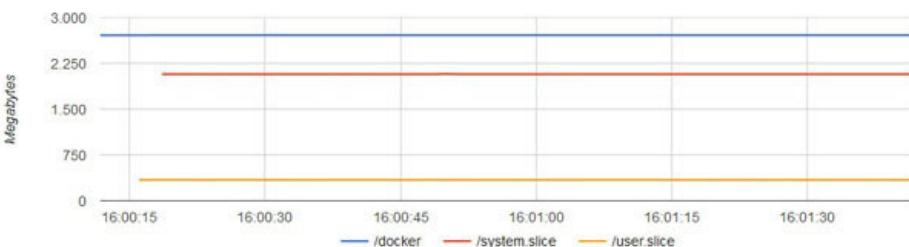


Figure 10.17: Usage of CPU and memory per container

The number of details provided by this API should be enough for many of the process monitoring and CPU usage needs.

Performance monitoring with Dive

A dive is a tool that allows you to explore the images of Docker, the content of one of the layers that make up the layer image, as well as the sizes and percentage of image efficiency from the perspective of size, a feature that would allow reducing the size of the images.

You can find the GitHub repository in <https://github.com/wagoodman/dive>.

Among the main features we can highlight:

Show the contents of the Docker image layer by layer: When selecting a specific layer, the content of that layer will be displayed in combination with all the previous layers on the right.

Indicator of changes in each layer: The file tree displays files that have changed, updated, inserted, or removed. It is possible to adjust this indicator to display adjustments for a specific surface.

Get image efficiency. The lower left panel + displays basic information for each surface, as well as an unconventional metric that tells you if your picture is space-efficient. This can be due to file duplication across layers, file transfer to other layers. Both a percentage of punctuation and the total wasted file space are provided.

Rapid construction / analysis cycles: You can create a Docker image and perform an immediate analysis with a single command: `dive build -t`.

For executing this tool, we can download the following image from the Docker hub.

```
$ docker pull quay.io/wagoodman/dive
```

To execute it, it is necessary to indicate the Docker socket file together with the identifier of the image we want to analyze.

```
$ docker run --rm -it \
```

```
-v /var/run/docker.sock:/var/run/docker.sockwagoodman/dive:latest
```

In the following screenshot we can see the output of the previous command:

Figure 10.18: Executing Dive container

The next step is executing the previous container with a specific image identifier:

```
$ docker run --rm -it \
-v /var/run/docker.sock:/var/run/docker.sock \
wagoodman/dive:latest
```

In the following screenshot we can see the output of the previous command:

Figure 10.19: Executing Dive container with specific image identifier

When executing Dive container with specific image identifier, we can obtain the metadata and layers of that image:

Figure 10.20: Layer details inside the image

When selecting a specific layer inside an image, we can see the layer details and the folder structure of this layer. Also, we can see information related to the command that is generating that layer, image size, potentially wasted space, and image efficiency score.

Container monitoring with Sysdig falco

From the perspective of monitoring docker containers with this tool, we can control the behavior of the containers. Sysdig Falco allows you to monitor all activity of containers, applications, and networks, as we would do with a combination of Unix tools such as Snort, tcpdump, htop, iftop, lsof and strace.

Sysdig Falco policies are a collection of rules that act directly on the flow of kernel system calls. These are the behaviors that Sysdig Falco can detect:

A shell that runs inside a container.

A process, in turn, generates another process with unexpected behavior.

Reading a confidential file, for example, the user file and passwords /etc/shadow.

A process is using a file that is not a device type in the /dev path, indicating a possible rootkit activity.

Behavior monitoring

Sysdig Falco focuses on the control at the level of behavior, which allows gaining visibility within the containers through the instrumentation of system calls. The call instrumentation of the system is completely transparent to the containers in execution, so it is not necessary to modify the code or images. The interception of system calls is done through a kernel module that is compiled dynamically.

When any abnormal activity is detected, a security event is emitted, such as an alert. The conditions that trigger the alert are defined by its policy, a collection of rules whose syntax is easy and works similar to calls to tcpdump.

We can combine different conditions from various sources such as events, metadata, and process information:

System call events: evt.type = listen+-, evt.type = mkdir, evt.type = setns

Docker metadata: container.image, container.privileged, container.name

Process tree information: proc.pname, proc.cmdline

For example, we can create a Falco rule that detects any connection socket outside our listening context, when:

The image of the container is nginx.

The listening process inside that container is nginx.

The syntax for creating this rule can be the following:

```
condition: evt.type in (accept,listen) and (container.image!=myregistry/nginx or proc.name!=nginx)
```

In the following URL, we have available more use cases for monitoring containers with Sysdigfalco.

<https://www.katacoda.com/sysdig/>

Wordpress container monitoring

In this example, we will see how to monitor a container in a Wordpress implementation using Docker. We will perform a basic exploration of containers and processes in containers, CPU monitoring, network, and I/O files.

In this scenario, we will see a Wordpress implementation in two instances with an HAProxy load balancer and a MySQL database. All services will be implemented using Docker containers.

The first thing we can do is start 4 containers, one for MySQL, 2 for Wordpress, and another for a nginx proxy.

This is the command for executing a container with MySQL:

```
$ docker run --name mysql -v /data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD='password' -d mysql
```

This is the commands for executing 2 containers with Wordpress:

```
$ docker run --name wp1 -e VIRTUAL_HOST=wp --link mysql:mysql -d wordpress
```

```
$ docker run --name wp2 -e VIRTUAL_HOST=wp --link mysql:mysql -d wordpress
```

This is the command for executing a container with proxy nginx:

```
$ docker run --name proxy -p 80:80 -e DEFAULT_HOST=wp -v /var/run/docker.sock:/tmp/docker.sock:ro -d jwilder/nginx-proxy:alpine
```

In the following screenshot we can see the output of the previous commands:

Figure 10.21: Executing containers MySQL, Wordpress, and nginx

Next, we are going to review how Sysdig works by capturing system calls that are made within the Linux kernel. This gives Sysdig visibility of how applications work within containers. Sysdig can operate with both real-time data and previously captured data.

Launching Sysdig container

Sysdig can be started as a container in the host Docker. For this, we can use the sysdig/sysdig image that we can find in the public repository in the Docker hub:

<https://hub.docker.com/r/sysdig/sysdig>

```
$ docker run -it --rm --name=sysdig --privileged=true \
--volume=/var/run/docker.sock:/host/var/run/docker.sock \
--volume=/dev:/host/dev \
--volume=/proc:/host/proc:ro \
--volume=/boot:/host/boot:ro \
--volume=/lib/modules:/host/lib/modules:ro \
--volume=/usr:/host/usr:ro \
sysdig/sysdig
```

Once we have Sysdig running, it will capture and display information related to system calls and events. This includes information such as the CPU where it was run, the name of the process, the thread identification and the type of event.

Sysdig offers a series of preconfigured filters that allow you to filter system-specific calls similar we can do with tcpdump command. The following command lists all system calls related to files and input/output operations in the path /etc .

```
$ sysdig "fd.name contains /etc"
```

With the following command, all the containers in execution are obtained. Internally, what it does is identify system calls executed in different user namespaces and inform the container name for each namespace.

```
$ sysdig -c lscontainers
```

In the following screenshot we can see the output of the previous command:

Figure 10.22: Inspecting containers with Sysdig

In the following screenshot, we can see examples of sysdig command execution:

Figure 10.23: Command execution with sysdig

You can get the full list of events by running the sysdig -L command:

Figure 10.24: List of Sysdig events

Regarding the output format of the events, by default, Sysdig returns the information of each event captured with the following format:

% evt.num% evt.outputtime% evt.cpu% proc.name (% thread.tid)% evt.dir% evt.type% evt.info

evt.num is the event number

evt.time is the timestamp of the event

evt.cpu is the CPU number where the event was captured

proc.name is the name of the process that generated the event

thread.tid is the identifier of the thread that generated the event, which corresponds to the PID for single-threaded processes

evt.dir is the address of the event-> for inbound events and

evt.type is the type of the event, (open, read)

evt.info is the event argument list

Sysdig filters

Sysdig allows filtering using information from different sources like system calls and events, file descriptors, process name, UID, PID. Also, it provides filters available to inform about system status, application protocols such as HTTP or Memcached, CPU usage, I/O activity, log logs, and network activity.

The following command allows you to check the filters we have available:

```
$ sysdig -cl
```

In the following screenshot we can see the output of the previous command:

Figure 10.25: List of sysdig events

We could obtain processes and CPU usage of running containers with the following command:

```
$ sysdig -pc -c topprocs_cpu
```

In the following screenshot we can see the output of the previous command:

Figure 10.26: CPU usage per container

With the following command, we can get processes that make more use of I/O:

```
$ sysdig -pc -c topprocs_file
```

Figure 10.27: Processes that make more use of I/O

One of the most difficult things to do when it comes to solving problems of Linux systems in general and of container-based infrastructure, in particular, is to observe the data that processes and containers exchange, both at the network level and at the file level.

For example, when trying to inspect network traffic, we could use tools like tcpdump. We can also filter by container name. With the httplog command, we can see the HTTP requests to each of the Wordpress servers.

```
$ sysdig -pc -c httplog
```

The following command will show all system calls executed by the Apache and MySQL processes, filtering only the calls that interest us:

```
$ sysdig -v -A -s 2048 "(proc.name=apache2 or proc.name=mysql) and evt.type!=gettmeofday and evt.type!=switch and evt.type!=iogetevents and evt.type!=futex and evt.type!=clock_gettime and evt.type!=epoll_wait and evt.type!=getsockopt and evt.type!=wait4 and evt.type!=select and evt.type!=semop"
```

In the following screenshot we can see the output of the previous command:

Figure 10.28: System calls executed by the Apache and MySQL processes

In the following URL we can find more examples of using Sysdig:

<https://www.sysdig.org/wiki/sysdig-examples>

Sysdig can also be used from the console interface called csysdig, a user interface developed with the ncurses library.

Csysdig as a tool to analyze system calls

Csysdig can be understood as a combination of popular tools such as strace, tcpdump, lsof, etc. with a user interface similar to the htop command for Linux that is designed to monitor and debug containers. On top of that, Csysdig works with real-time and historical data allowing having several views available for different scenarios.

In the following screenshot we can see the execution of csysdig:

Figure 10.29: Csysdig execution

This is the aspect of the user interface of the csysdig command. The view menu can be accessed using F2. In each view, you can see different columns explained while navigating between views. For example, in the 'Containers' view, you can see CPU, a number of processes, memory, file I/O, or network by container name and ID. You can always select any of the rows and start filtering by that container.

For more information about csysdig, we recommend you check the URL <https://sysdig.com/blog/csysdig-explained-visually>.

Conclusion

In this chapter, we have reviewed how the container gives a lot of information about CPU, processes, threads, memory, and network information for each container. In this chapter, the reader has learned some open source tools available for Docker container monitoring and others that allow filtering using information from different sources like system calls and events that occur in the container.

In the next chapter, we will review open-source tools available for Docker container administration such as rancher and portainer.io.

Questions

Which is the path where logs are located on the Docker host by default?

Which command allows you to obtain statistics for one or more containers in execution and see information like the use of CPU, memory, I/O operations at the network level?

Which command allows you to see what life cycle events are happening in real-time inside the container?

Which tool is one of the best useful tools that enable container-oriented performance monitoring and runs as a daemon process that collects performance data in running containers?

Which tool allows you to monitor all activity of containers, applications, and networks, as we would do with a combination of Unix tools such as Snort, tcpdump, htop, iftop, lsof, and strace?

CHAPTER 11

Docker Container Administration

This chapter introduces some of the open-source tools available for Docker container administration, such as rancher and portainer.io.

Containers constitute a complete execution environment, which includes an application, its dependencies, libraries, binary files, and configuration necessary for execution, tied in a package. This is called containerization and helps to be abstracted from the application platform and its dependencies, differences in operational distributions, and the underlying infrastructure.

However, to move dockerized applications to production containers, appropriate management tools are required to ensure security, automation, orchestration, and administration. Today it is essential to define the way in which images and containers are deployed in different environments until they reach production, and Docker helps developers innovate more quickly.

Structure

Introducing container administration

Container administration with rancher

Container administration with portainer.io

Objectives

Knowing about container administration

Knowing about container administration with rancher

Knowing about container administration with portainer.io

Introducing container administration

It is important that organizations and developers consider the challenges associated with managing Docker environments and the need to implement business solutions that support effective management while deploying Docker containers, which must-have technology that allows us to successfully manage the problems of dispersion, compliance, and governance of the same containers.

The three stages of the container life cycle are:

Development: In the first stage, developers create and deploy Docker containers that include items such as application codes and libraries. Then they test the applications, correct errors, add functions or improvements, create new Docker images, and deploy them in new containers. This process continues until the required standards are met.

Application release: In the second stage, managers coordinate the automation of application environments that include Docker construction, testing, and deployment drivers.

IT operations: In the last stage, the containers are deployed in production and remain operational and available until they are dismantled. This is the stage in which the final challenges are critical: orchestration and governance, security, and container monitoring.

To harness the potential of Docker's benefits, developers and organizations need solutions designed to address five major container management challenges:

Lack of control: Developers need independence to create, implement and test application containers quickly. In contrast, the operations team needs control and governance to avoid excessive consumption of resources.

Cycle from rising to production: As changes in development increase, it is important to manage to maintain quality and safety.

Complexity of scale containers: The virtualized or cloud infrastructure does not disappear and will continue to coexist with the Docker infrastructure. The implementation of complete applications covering Docker and other infrastructures requires more advanced capabilities to orchestrate applications and optimally manage running environments.

Vulnerability protection and compliance: Because they include parts of the operating systems, Docker containers can integrate vulnerabilities such as Heartbleed and Ghost. The protection of the environment requires security in the host Docker layer, in containers, and in the images. The container update creates a new management paradigm that can change the tasks of operations to development.

Monitoring requirements: Docker environments require special monitoring capabilities, such as API-level integration with Docker and instrumentation built into the Docker image.

To take full advantage of Docker's benefits, organizations need the appropriate management and administration tools that allow them to manage the full life cycle of the Docker container and ensure the company's availability for both development and production environments.

Within the Docker ecosystem, we can find some interesting tools for developers to manage the process of managing images and containers safely. Among the main tools for container administration, we can highlight Rancher and Portainer.

Portainer is a user interface that allows you to manage different Docker environments (at the host level or at the cluster level with Swarm). This tool consists of a single container that can be run on any Docker engine, and it can be implemented as a Linux container or a native Windows container.

Rancher <http://rancher.com> is an open-source platform that runs on Docker and allows applications to be deployed in container-based. The

platform has a Hosts section to visually manage the machines or instances of different clouds, whether AWS (Amazon), Azure (Microsoft), or Digitalocean. It can be said that it is a visual console, for instance, management.

There are other solutions for administration, such as the Dockstation. Dockstation <https://dockstation.io> is an application as a user interfaces for container management in Docker. First, you create your project and establish a dockercompose.yml file or a docker execution command that allows you to start the application. For more information about installation and use, you can access the public repository in GitHub <https://github.com/DockStation/dockstation>.

Among the main features, we can highlight that it allows seeing the resource consumption of each container, as well as to monitor the state of the containers in execution, indicating the usage of resources, CPU, memory, and network.

In the next section, we will learn to use rancher to orchestrate our container stacks in Docker.

Container administration with rancher

Rancher is a platform that allows you to manage containers and stacks of containers on remote servers, instead of entering the server house and managing your containers, you can do everything from one place as well as take advantage of all the features that Rancher offers. In the followingURL, you can see the installation requirements:

<https://rancher.com/docs/rancher/v2.x/en/installation/requirements>

To install rancher on a server with a Docker container, simply run the command:

```
$ docker run -d --restart=unless-stopped -p 8080:8080 rancher/server:stable
```

The previous command downloads the official image of Rancher, and we will have Rancher server running, and you can access the panel interface on port 8080 .

Among the main advantages that Rancher offers, we can highlight:

It allows you to create as many environments as you need and manage users and roles for different environments.

It allows you to select the container orchestrator from several such as Cattle, Mesos, Kubernetes and Docker Swarm.

There is a public catalog called Rancher community where the community can contribute to its applications.

The platform has a Hosts section to visually manage the machines or instances of different clouds, either AWS (Amazon), Azure (Microsoft), and Digitalocean. You can say that it is a visual console, for instance, management.

One of the options offered by Rancher is the possibility of adding a host for deploying containers and stacks:

Hosts: Add Host

Custom (selected)

AMAZON EC2

Azure

DigitalOcean

packet

Manage available machine drivers

- 1 Start up a Linux machine somewhere and install a supported version of Docker on it.
- 2 Make sure any security groups or firewalls allow traffic:
 - From and To all other hosts on UDP ports 500 and 4500 (for IPsec networking)
- 3 Optional: Add labels to be applied to the host.

Figure 11.1: Add host in Rancher interface

The application has a simple interface; on the one hand, the hosts can create containers and start applications made by the containers. Rancher manages agents to establish communication between him and his hosts; that's why we must install that agent. It is a very simple process. Simply add the host's from the Rancher console, following these steps:

Within the menu, we choose the options Infrastructure | Hosts

We follow the steps marked by the wizard to install the agent on the host

We execute the command on the host that we want Rancher to manage

In the following screenshot, we can see the steps for adding a host with the command we need to execute for registering the host:

1 Start up a Linux machine somewhere and install a **supported version of Docker** on it.

2 Make sure any security groups or firewalls allow traffic:
◦ From and To all other hosts on **UDP** ports **500** and **4500** (for IPsec networking)

3 Optional: Add labels to be applied to the host.
+ Add Label

4 Specify the public IP that should be registered for this host. If left empty, Rancher will auto-detect the IP to use. This generally works for machines with unique public IPs, but will not work if the machine is behind a firewall/NAT or if it is the same machine that is running the **rancher/server** container.
e.g. 1.2.3.4

5 Copy, paste, and run the command below to register the host with Rancher:

```
sudo docker run --rm --privileged -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/rancher:/var/lib/rancher rancher/agent:v1.2.11 https://2886795319-8080-cykoria03.environments.katacoda.com/v1/scripts/54458B0E4F5E7FC55C78;1546214400000:GbqLiVowScu5Xi2iQJDclMykT8
```

Copy to Clipboard

Figure 11.2: Registering host in Rancher interface

After executing this command on your new host with Linux to add the host, once you run the script, you must wait a few minutes to see the host in Rancher:

The screenshot shows the Rancher web interface. At the top, there is a navigation bar with icons for cattle, a house labeled "Default", "STACKS", "CATALOG", "INFRASTRUCTURE", "ADMIN" (with a red exclamation mark), and "API". Below the navigation bar, the main area is titled "Hosts" with a "Add Host" button. The "ACTIVE" tab is selected, showing a host named "master". The host details include its IP (195.154.79.10), port (18.09.7), operating system (Ubuntu 16.04.6 LTS (4.4.0)), and hardware specifications (2x2.1 GHz CPU, 1.95 GiB RAM, 44.1 GiB storage). Below the host details, there are two sections for stacks: "Stack: healthcheck" and "Stack: ipsec". The "Stack: healthcheck" section lists a single service "healthcheck-1" with IP 10.42.76.44. The "Stack: ipsec" section lists services "cni-driver-1" (IP None) and "ipsec-1" (IP 10.42.135.94), along with two sidekick services.

Figure 11.3: Information about host in Rancher interface

We could also configure and add different development environments using some environment templates depending on the orchestration platform we are using:

The screenshot shows the "Add Environment" page in the Rancher interface. It has fields for "Name" (e.g. lab) and "Description" (e.g. Environment for developer experimentation). Below these, there is a section for "Environment Template" with five options: Cattle (selected with a checkmark), Kubernetes, Mesos, Swarm, and Windows. Below the template selection, it says "Orchestration: Cattle", "Framework: Network Services, Scheduler, Healthcheck Service", and "Networking: Rancher IPsec".

Figure 11.4: Environment templates in Rancher interface

In this screenshot we can see the templates available by default offered by Rancher:

Environment Templates

Add Template

An environment template allows users to define a different combination of infrastructure services to be deployed.

The infrastructure services includes but not limited to container orchestration (i.e. cattle, kubernetes, mesos, swarm, networking) or rancher services (i.e healthcheck, dns, metadata, scheduling, service discovery and storage)

Name	Description	Stacks	Public	
Cattle	Default Cattle template	network-services, ipsec, scheduler, healthcheck	✓	
Kubernetes	Default Kubernetes template	kubernetes, network-services, ipsec, healthcheck	✓	
Mesos	Default Mesos template	mesos, network-services, ipsec, scheduler, healthcheck	✓	
Swarm	Default Swarm template	portainer, swarm, network-services, ipsec, scheduler, healthcheck	✓	
Windows	Experimental Windows template	windows, windows-network-services	✓	

Figure 11.5: Environment templates in Rancher interface

Another of the most important aspects of Rancher is its catalog of applications. This catalog is public, and the open-source community can contribute its applications to all Rancher community users. It also offers the possibility of having a private application catalog.

In this screenshot you can see the applications catalog available in Rancher interface:

The screenshot shows the Rancher Applications Catalog interface. At the top, there is a navigation bar with icons for Home, Default, Stacks, Catalog (selected), Infrastructure, Admin, and API. Below the navigation bar, the title "Catalog: All" is displayed. On the right side of the catalog page, there are search and filter options: "Search..." and "Category: All". A "Manage" button is also present. The catalog displays four application cards:

- Alfresco**: An ECM and BPM platform. View Details
- Alibaba Cloud DNS**: Rancher External DNS service powered by Alibaba Cloud. View Details
- Apache Guacamole**: Apache Guacamole is a clientless remote desktop gateway. It supports standard protocols like VNC, RDP, and SSH. View Details
- Apache Kafka**: Kafka cluster. View Details

Figure 11.6: Applicationscatalog in Rancher interface

Rancher provides a web interface to control containers. The dashboard shows starting, stopped, and running containers:

The screenshot shows the Rancher Container dashboard. At the top, there are buttons for 'Add Container' and 'Show System'. Below is a search bar and navigation controls for '25 Items'. The main table lists the following containers:

State	Name	IP Address	Host	Image	Command
Starting	coredns	None	master	sha256:eb516548c180f8a6e023	None
Starting	coredns	None	master	sha256:eb516548c180f8a6e023	None
Stopped	coredns-fb8b8dccf-ndxkx	10.42.226.169	master	k8s.gcr.io/pause:3.1	None
Stopped	coredns-fb8b8dccf-zcf7x	10.42.7.135	master	k8s.gcr.io/pause:3.1	None
Stopped	etcd	None	master	sha256:2c4adeb21b4ff8ed3309c	None
Stopped	etcd-master	None	master	k8s.gcr.io/pause:3.1	None
Running	healthcheck-healthcheck-1	10.42.76.44	master	rancher/healthcheck:v0.3.8	healthcheck,--metadata-addr...
Running	ipsec-cni-driver-1	None	master	rancher/net:v0.13.17	start-cni-driver.sh
Running	ipsec-ipsec-1	10.42.135.94	master	rancher/net:holder	sh-c.echo Refer to router side...

Figure 11.7: Container dashboard in Rancher interface

By clicking Add Container, you will be redirected to a page where you can set the container run parameters:

The screenshot shows the 'Add Container' configuration dialog. It includes fields for Name (nginx), Description (nginx), Select Image* (nginx), and a checkbox for Always pull image before creating. Below these are fields for mapping ports: Public Host Port (80) and Private Container Port (80), with a dropdown for Protocol (TCP). A 'Port Map' button is also present.

Figure 11.8: Container dashboard in Rancher interface

The Containers section lists all your running containers. You can open a shell into a container, stop, restart, and delete the container and other options related to logs and clone the container.

In this screenshot you can see the containers in running state:

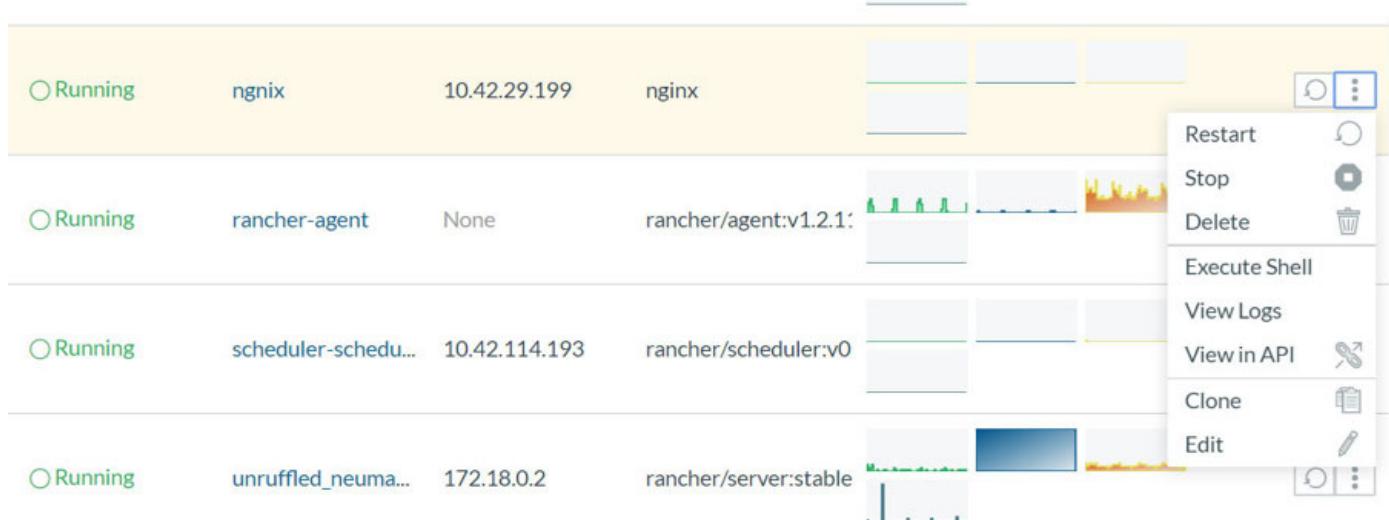


Figure 11.9: Running containers in Rancher interface

In this section, we have reviewed how you can deploy a container from the container dashboard and see the state of each one container from the Rancher interface.

Deploying Kubernetes using Rancher

With Rancher, you can initialize multiple clusters with one single central place to manage them. The Rancher control plane is deployed as a Docker Container.

To start Rancher, run the command:

```
$ docker run -d -p 80:80 -p 443:443 --name=rancher rancher/rancher:stable
```

If we execute a docker ps command, we can see the rancher container in execution:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
STATUS   PORTS      NAMES
9f424719a637        rancher/rancher:stable   "entrypoint.sh"   6 seconds ago   Up 3 seconds   0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
rancher
```

Rancher may take one or two minutes to start. You can view the boot process with the command:

```
$ docker logs rancher
```

Once started, the first step is to configure a password for the login admin of the cluster. For a random password, have Rancher generate the password for you.

In this screenshot you can see the welcome page to Rancher for setting the password to the admin user:

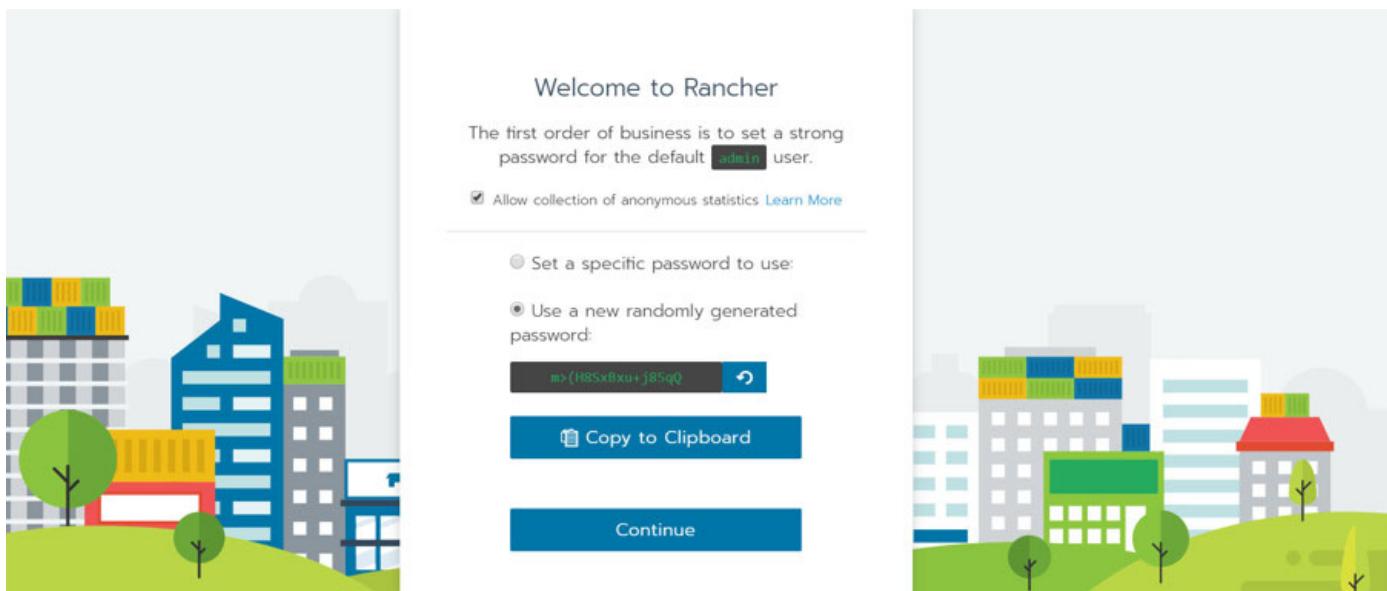


Figure 11.10: Welcome page to Rancher

You can see the admin user has been created in the Users section:

State	Name	ID	Local Username
Active	Default Admin (admin) Local User	user-xcnpm	admin

Figure 11.11: Users section in Rancher interface

In the next step, you'll create a cluster to configure Kubernetes:



From existing nodes (Custom)

Create a new Kubernetes cluster using RKE, out of existing bare-metal servers or virtual machines.



Import an existing cluster

Import an existing Kubernetes cluster. The provider that created it will continue to manage the provisioning and configuration of the cluster.

With RKE and new nodes in an infrastructure provider



Amazon EC2



Azure



DigitalOcean



Linode



vSphere

Figure 11.12: Add Cluster-Select Cluster Type

In this scenario, we'll deploy an on-premise solution. To begin the installation, click the Add Cluster button and select the cluster type of custom:



Add Cluster - Custom

Cluster Name * Add a Description

Member Roles
Control who has access to the cluster and what permission they have to change it.

Labels & Annotations
Configure labels and annotations for the cluster. None

Cluster Options

[Edit as a Form](#)

Figure 11.13: Add Cluster-Custom

In the cluster configuration you can edit and customize the Kubernetes cluster options:

[Copy to Clipboard](#)

[Read from a file](#)

```
1 #
2 # Cluster Config
3 #
4 docker_root_dir: /var/lib/docker
5 enable_cluster_alerting: false
6 enable_cluster_monitoring: false
7 enable_network_policy: false
8 local_cluster_auth_endpoint:
9   enabled: true
10 name: k8s-katacoda-staging
11 #
12 # Rancher Config
13 #
14 rancher_kubernetes_engine_config:
15   addon_job_timeout: 30
16   authentication:
17     strategy: x509
18   ignore_docker_version: true
19 #
```

Figure 11.14: Cluster file configuration

The next step is to deploy Kubernetes; in this case, we'll start by configuring a single node instance of Kubernetes. A single node instance has etcd , the Kubernetes Control Plane and a Kubernetes node to all run on the same machine. Within production, this would be deployment onto multiple nodes; however, a single node is a great starting place for testing and experimenting.

By selecting the etcd and control plane boxes, the command to initialize the cluster at the top will change. This command will deploy the correct configuration for our single node cluster.

1 Node Options
Choose what roles the node will have in the cluster

Node Role

etcd Control Plane Worker

Show advanced options

2 Run this command on one or more existing machines already running a supported version of Docker.

```
sudo docker run -d --privileged --restart=unless-stopped --net=host -v /etc/kubernetes:/etc/kubernetes -v /var/run:/var/run rancher/rancher-agent:v2.3.2 --server https://2886795332-80-cykoria01.environments.katacoda.com --token hgj6qvlgbm5grnlz4jj5qs1vtkdkjh7x67j2wrzp6vxj95kkrmvmt --ca-checksum c02cc0b5a7203f0c0c4b29bcd14af55ca1c2ed73949c9b974746ecc3397a6cd3 --etcd --controlplane --worker
```

Figure 11.15: Cluster node configuration

When you're ok with the configuration, run the command in the Terminal window. Rancher has a helpful Copy to Clipboard button to make this easier.

After running the command, the Rancher dashboard should report 1 new node is being registered:

This screenshot shows the 'Nodes' section of the Rancher UI. At the top, a message states: 'This cluster is currently Provisioning, areas that interact directly with it will not be available until the API is ready.' Below this, another message says 'Waiting for etcd and controlplane nodes to be registered'. The main table lists one node: 'master' with IP '172.17.0.68'. The 'State' column shows 'Registering', and the 'Roles' column shows 'Worker'. Other columns include 'Version' (n/a), 'CPU' (n/a), 'RAM' (n/a), and 'Pods' (n/a). A note at the bottom says 'Waiting to register with Kubernetes'.

Figure 11.16: Registering a master node in the cluster

Rancher is now starting all the components of Kubernetes. This will take a couple of minutes. Afterward, you will have a fully functional Kubernetes cluster. Within the UI, you can select the newly deployed cluster to view the details, usage, and status.

In this screenshot you can see the cluster state in Rancher interface:

This screenshot shows the 'Clusters' section of the Rancher UI. At the top, a message says 'Waiting for etcd and controlplane nodes to be registered'. The main table lists one cluster: 'k8s-katacoda-staging'. The 'State' column shows 'Provisioning', and the 'Provider' column shows 'Custom'. Other columns include 'Nodes' (1), 'CPU' (n/a), and 'RAM' (n/a). A note at the bottom says 'Waiting for etcd and controlplane nodes to be registered'.

Figure 11.17: Cluster state in Rancher interface

In the details of the cluster, you can select Launch Kubectl. This provides you with an interface to manage and control your cluster. Try running kubectl get nodes to see the nodes available in the cluster.

In the next section, we will review Portainer as a lightweight and open source user interface for Docker. With Portainer, you can extract images, add containers, add networks, and much more.

Container administration with portainer.io

Portainer is an open-source web tool that runs itself as a container. This application will allow us to manage our Docker containers very easily and intuitively through a graphical interface. Through a web interface, an administrator can have a clearer overview of the containers he is running and facilitate his management.

In the Docker hub repository, we can find the official image for deploying this tool as a Docker container:

<https://hub.docker.com/r/portainer/portainer> .

The easiest way is by using the following command to start a container with the Portainer application on port 9000 from which we can view the containers in execution on our Docker host.

In order to Portainer can manage the local Docker server, you must include -v /var/run/docker.sock:/var/run/docker.sock in the docker run command. Then, we proceed to download and start the container:

```
$ docker run -d -p 9000:9000 --name portainer --restart always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer
```

The docker run command options are the usual ones:

-d for running the container as a background process.

--name portainer to give the container a name.

-p 9090:9000 to connect port 9090 of the host with port 9000 exposed in the container.

-v portainer-data:/data to persist the Portainer configuration on a volume outside the container.

-v /var/run/docker.sock:/var/run/docker.sock mounts the UNIX sock in the container.

The previous command what it does is executing the container and listen on port 9000 . Therefore, to access it, we must simply go to <http://localhost:9000> in our browser. portainer/portainer is the Docker hub repository from where we download the Portainer image.

In the following screenshot we can see the output of the previous command:

```
# docker run -d -p 9000:9000 --name portainer \
> -v /var/run/docker.sock:/var/run/docker.sock \
> -v portainer_data:/data --restart=always portainer/portainer
Unable to find image 'portainer/portainer:latest' locally
latest: Pulling from portainer/portainer
d1e017099d17: Pull complete
8f8668d9390b: Pull complete
Digest: sha256:339b6486297050179418c886272f3262794b54513008a7f8e747c5e8f330338d
Status: Downloaded newer image for portainer/portainer:latest
d0c45bc24eb1f6d0bbd087f8c125ac856d5b9ba8463007fdfeffff9689b53efc
```

Figure 11.18: Starting Portainer container

Another way to execute it is by using the docker-compose up -d command from the following docker-compose.yml file:

version: '2'

services:

portainer:

image: portainer/portainer

ports:

- "9500:9000"

command: -H unix:///var/run/docker.sock

volumes:

- /var/run/docker.sock:/var/run/docker.sock
- portainer_data:/data

volumes:

portainer_data:

The first time you access the Portainer URL, you must enter the admin user password. When accessing via the web interface for the first time, you will be asked to configure the administrator user password, as shown in the image:



Please create the initial administrator user.

Username	admin
Password	<input type="password"/>
Confirm password	<input type="password"/> ×

× The password must be at least 8 characters long

Create user

Figure 11.19: Creating an admin user

Once the administrator user has been created with a valid password for the password policy, you will go to the next window, where you will ask if we want to raise it locally or remotely. In this screenshot we can see we have selected local installation to handle the containers:



Connect Portainer to the Docker environment you want to manage.

<input checked="" type="radio"/> Local Manage the local Docker environment	<input type="radio"/> Remote Manage a remote Docker environment	<input type="radio"/> Agent Connect to a Portainer agent	<input type="radio"/> Azure Connect to Microsoft Azure ACI
---	--	---	---

Information

Manage the Docker environment where Portainer is running.

1 Ensure that you have started the Portainer container with the following Docker flag:
`-v "/var/run/docker.sock:/var/run/docker.sock"` (Linux).
 or
`-v '\\.\pipe\docker_engine:\\.\pipe\docker_engine` (Windows).

Connect

Figure 11.20: Local installation for managing Portainer

To display information about the containers (images, volumes, etc.) in Docker, Portainer needs to connect via API to the host on which Docker is running.

With the following option, we could execute Portainer remotely:

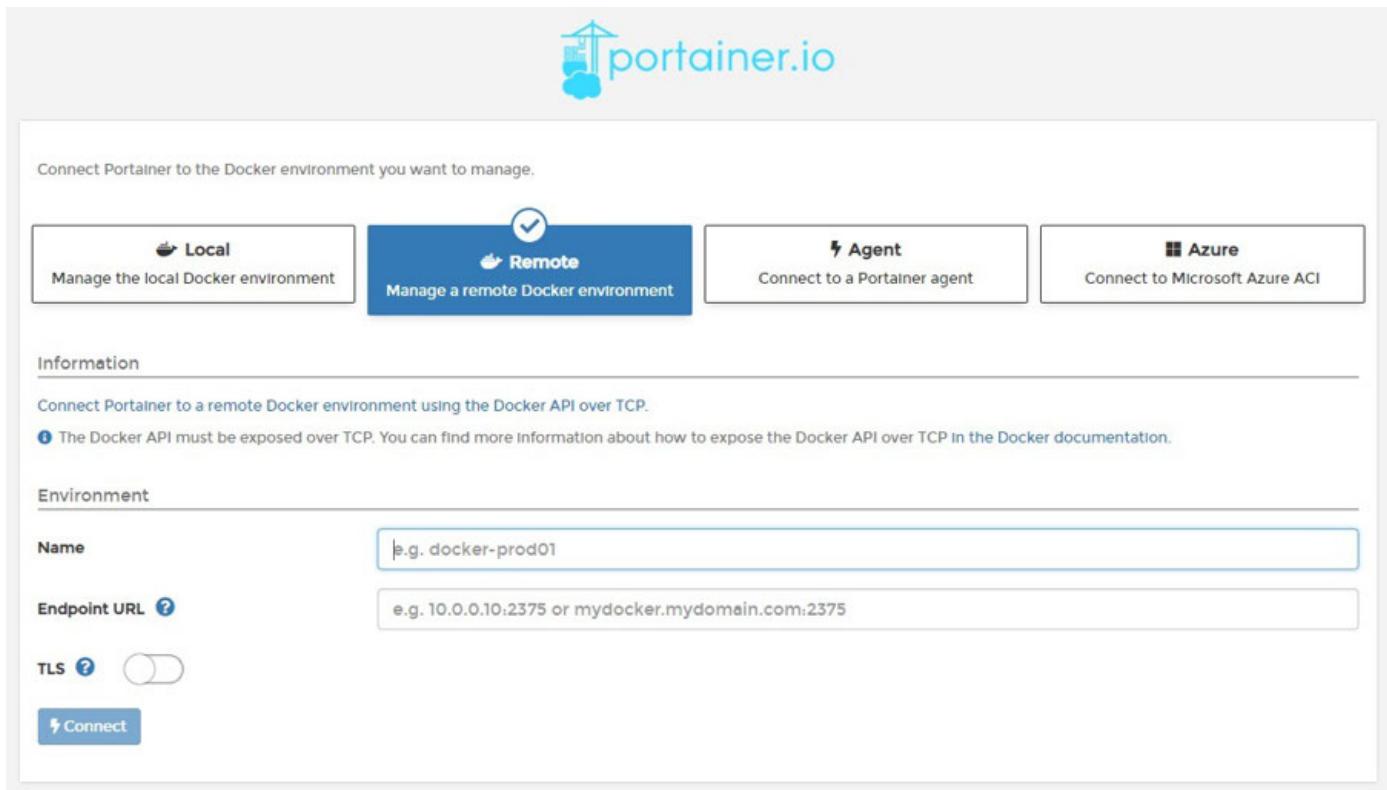


Figure 11.21: Remote execution for managing Portainer

If we press the Connect button, we can see the status of our dockerversion, CPU, memory, and so on. We see in the image that we only have one Docker node, which is the primary.

The Portainer interface gives us the information of its volumes, images, and containers, both those that are stopped and those that are running:

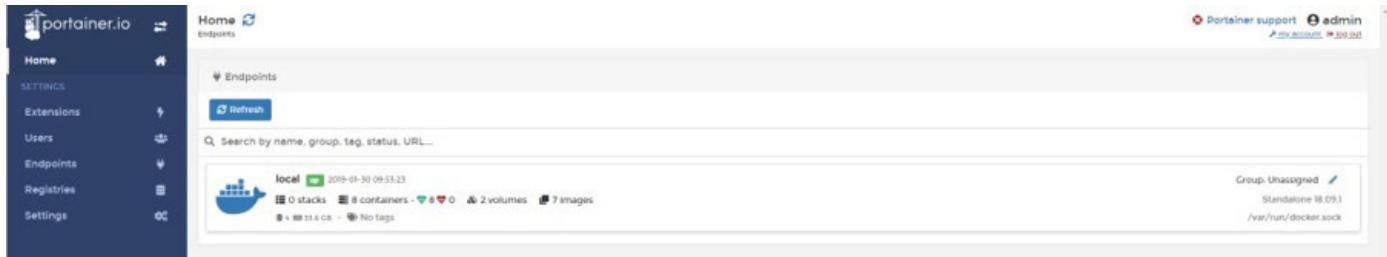


Figure 11.22: Portainer interface

The Dashboard is the main page of our Portainer instance. It shows a visual summary of our Docker system: the total number of containers, images, networks, or volumes. This section is very useful to show us globally what is the current status of Docker in our machine.

If we click on the server, we see all stacks, containers, images, volumes, and networks that have been created in the Docker instance:

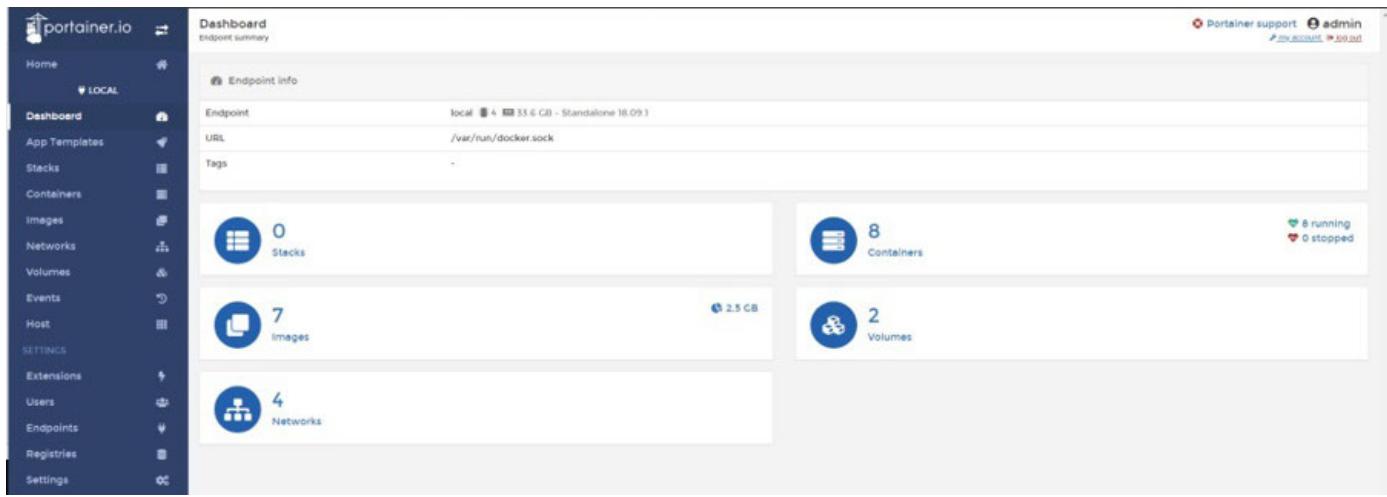


Figure 11.23: Portainer Dashboard

The Containers menu will show us the list of all our containers, and we will be able to execute several of the typical instructions that we usually execute through the command line, such as starting, stopping, or eliminating them.

In the following screenshot we can see the container list in Portainer interface:

Container list									
Containers									
	<input type="checkbox"/> Name	State	Quick actions	Stack	Image	Created	IP Address	Published Ports	Ownership
<input type="checkbox"/>	portainer			-	portainer/portainer	2019-01-30 09:51:42	172.17.0.8	9000:9000	administrators
<input type="checkbox"/>	traefik_proxy			-	traefik	2019-01-28 14:20:18	172.18.0.2	80:80 8100:8080	administrators
<input type="checkbox"/>	telegraf			-	svtalegraf/latest	2019-01-24 17:14:34	172.17.0.7	-	administrators
<input type="checkbox"/>	[REDACTED]			-	tomcat7war	2019-01-23 16:03:07	172.17.0.3	9105:8009 9003:8080	administrators
<input type="checkbox"/>	[REDACTED]			-	tomcat7war	2019-01-23 16:02:07	172.17.0.2	9104:8009 9004:8080	administrators
<input type="checkbox"/>	[REDACTED]			-	tomcat7war	2019-01-23 15:56:06	172.17.0.6	9105:8009 9005:8080	administrators
<input type="checkbox"/>	[REDACTED]			-	tomcat7war	2019-01-23 15:49:07	172.17.0.5	9102:8009 9002:8080	administrators
<input type="checkbox"/>	[REDACTED]			-	tomcat7war	2019-01-23 15:40:07	172.17.0.4	9001:8080 9101:8009	administrators

Figure 11.24: Container list in Portainer interface

We can also see the details of the container itself. If we click on the name of a container, then we can know its information. The Container details section allows us to perform some operations over the container:

Executing common operations such as stop, pause, kill, or delete the container.

See container information (docker inspect).

Create a new image from the same container and add it to a record (docker commit).

See container logs (docker logs)

See container statistics (docker stats).

Enter the container, choosing the shell or the user (docker exec).

Connect/disconnect the container with a network (docker network connect).

In the following screenshot we can see the Container details in Portainer interface:

Container details

Containers > nginx-web

Portainer support admin
[my account](#) [log out](#)

The screenshot shows the 'Container details' page for a container named 'nginx-web'. At the top, there's a row of quick actions: Start (green), Stop (red), Kill (orange), Restart (blue), Pause (yellow), Resume (green), Remove (red), Recreate (blue), and Duplicate/Edit (blue). Below this is a section titled 'Container status' with the following information:

ID	e2b1b62ee25363a264af65eef72f68cb416066d5de219b1d8c3ea830957c335e
Name	nginx-web Edit
IP address	172.17.0.2
Status	Running for 6 minutes
Created	2019-11-13 16:15:20
Start time	2019-11-13 16:15:20

At the bottom of this section are links for Logs, Inspect, Stats, Console, and Attach.

Figure 11.25: Container details in Portainer interface

If you look at the Quick Actions , we will see the following icons from left to the right:

Logs: Allows us to see, in real-time, the logs container.

Inspect: Give us all the information on the container.

Stats: Shows the statistics of the container (memory usage, CPU, network, and processes).

Console: Gives us access to the container console.

In the Logs section, we can see container logs. In the following screenshot we can see this section in Portainer interface:

The screenshot shows the 'Log viewer settings' section. It includes the following controls:

- Auto-refresh logs: A toggle switch that is turned on.
- Wrap lines: A toggle switch that is turned on.
- Display timestamps: A toggle switch that is turned off.
- Fetch: A dropdown menu set to 'All logs'.
- Search: An input field labeled 'Filter...'.
- Lines: An input field set to '100'.
- Actions: Buttons for 'Copy', 'Copy selected lines', and 'Unselect'.

Below these settings, the log output is displayed in a scrollable area:

```
* directory[/opt/gitlab/sv/grafana/control] action create (up to date)
* link[/opt/gitlab/init/grafana] action create (up to date)
* file[/opt/gitlab/sv/grafana/down] action delete (up to date)
* directory[/opt/gitlab/service] action create (up to date)
* link[/opt/gitlab/service/grafana] action create (up to date)
* ruby_block[wait for grafana service socket] action run (skipped due to not_if)
- execute the ruby block reload log service
```

Figure 11.26: Log details in Portainer interface

On Inspect section we can see the data of the container, simulating the docker inspect command:

Container inspect

Containers > nginx > Inspect

Portainer support admin
my account log out

Inspect

Tree Text

▼ 9c52919fc46953034a7b54ba15520a816b62e875f4c1b66967841066c9e550d7:
 AppArmorProfile: docker-default
 Args: [-g, daemon off]
 Config: { ArgsEscaped: true, AttachStderr: false, AttachStdin: false, Attach
 Created: 2019-11-13T16:15:34.455100998Z
 Driver: overlay2
 ExecIDs:
 GraphDriver: { Data: [object Object], Name: overlay2 }
 HostConfig: { AutoRemove: false, Binds: 21e33b5cd62877c370d2a7a897257142f
 HostnamePath: /var/lib/docker/containers/9c52919fc46953034a7b54ba15520a816b62e875f4c1b66967841066c9e550d7/hostname
 HostsPath: /var/lib/docker/containers/9c52919fc46953034a7b54ba15520a816b62e875f4c1b66967841066c9e550d7/hosts
 Id: 9c52919fc46953034a7b54ba15520a816b62e875f4c1b66967841066c9e550d7
 Image: sha256:540a289bab6cb1bf880086a9b803cf0c4cefe38cbb5cdefa199b69614525199f
 LogPath: /var/lib/docker/containers/9c52919fc46953034a7b54ba15520a816b62e875f4c1b66967841066c9e550d7/9c52919fc46953034a7b5
 4ba15520a816b62e875f4c1b66967841066c9e550d7-json.log
 MountLabel:
 Mounts: [[object Object], [object Object]]
 Name: /nginx
 } }

Figure 11.27: Docker inspect details in Portainer interface

On the Statistics section, we can see the graphics about the usage of CPU, memory, and network:

Container statistics

Containers > nginx > Stats

Portainer support admin
my account log out

About statistics

This view displays real-time statistics about the container nginx as well as a list of the running processes inside this container.

Refresh rate: 5s

Memory usage

CPU usage

Network usage (aggregate)

Processes

Search...

PID	USER	TIMF	COMMAND
-----	------	------	---------

Figure 11.28: Container statistics in Portainer interface

The Images section would correspond, if we were working with the terminal, to the docker images command:

The screenshot shows the 'Images' section of the Portainer interface. At the top, there are buttons for 'Remove', '+ Build a new image', 'Import', and 'Export'. Below is a search bar and a table with the following columns: 'Id', 'Tags', 'Size', and 'Created'. The table contains four rows of image entries:

Id	Tags	Size	Created
sha256:f6e8af4562c14ab06a2c9f3698e39e...	Unused dockersamples/examplevotingapp_vote:<none>	83.6 MB	2017-01-11 02:54:06
sha256:2b1e6048c5398e19b011fc1b67c2d1...	Unused dockersamples/examplevotingapp_worker:<none>	961.9 MB	2017-04-07 21:31:15
sha256:540a289bab6ccb1bf880086a9b803cf...	nginx:latest	126.2 MB	2019-10-23 02:26:03
sha256:36726735dc3c2c86ff47a937c72d53...	Unused postgres:<none>	206.3 MB	2019-10-17 06:40:54

Figure 11.29: Images section in Portainer interface

As we can see in the previous screenshot, the Portainer web interface for the images section offers us a series of advantages over the command line, among which we can highlight:

Sort the list by tags.

Sort the list by size.

Selection of one or more images through checkboxes.

Unused tag to show us images that are not currently using.

On the section Network list , we have the possibility to see the networks that we have already created. We can also remove them or add a new network through the interface.

In the following screenshot we can see the Network list in Portainer interface:

The screenshot shows the 'Network list' section of the Portainer interface. At the top, there are buttons for 'Remove' and '+ Add network'. Below is a search bar and a table with the following columns: 'Name', 'Stack', 'Scope', 'Driver', 'Attachable', 'Internal', 'IPAM Driver', 'IPAM Subnet', 'IPAM Gateway', and 'Ownership'. The table contains five rows of network entries:

Name	Stack	Scope	Driver	Attachable	Internal	IPAM Driver	IPAM Subnet	IPAM Gateway	Ownership
bridge	-	local	bridge	false	false	default	172.17.0.0/16	-	administrators
docker_gwbridge	-	local	bridge	false	false	default	172.19.0.0/16	172.19.0.1	administrators
host	-	local	host	false	false	default	-	-	administrators
ingress	-	swarm	overlay	false	false	default	10.255.0.0/16	10.255.0.1	administrators
none	-	local	null	false	false	default	-	-	administrators

Figure 11.30: Network list section in Portainer interface

In the Volumeslist section, we have the possibility to see the volumes that we have already created. We can also remove them or add a new volume through the web interface.

In the following screenshot we can see the Volumes list in Portainer interface:

Figure 11.31: Volume list section in Portainer interface

In the section called App Templates , we can find a lot of templates available to download and install:

Name	Description	Action
Portainer Agent	Manage all the resources in your Swarm cluster	<input type="button" value="Update"/> <input type="button" value="Delete"/>
OpenFaaS	Serverless functions made simple	<input type="button" value="Update"/> <input type="button" value="Delete"/>
IronFunctions	Open-source serverless computing platform	<input type="button" value="Update"/> <input type="button" value="Delete"/>

Figure 11.32: App Templates section in Portainer interface

If we activate the check button Show container templates , it will show more templates related to container images like Docker registry or MySQL.

In the following screenshot we can see the list of some container templates you can deploy with Portainer interface:

Show container templates

Search...

 Registry <small>container</small> Docker image registry	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> docker
 Nginx <small>container</small> High performance web server	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> webserver
 Httpd <small>container</small> Open-source HTTP server	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> webserver
 Caddy <small>container</small> HTTP/2 web server with automatic HTTPS	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> webserver
 MySQL <small>container</small> The most popular open-source database	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> database

Figure 11.33: ContainerTemplates in Portainer interface

To show how it works, we can download a predefined container template from Add Templates .

In the example, we will use the template to install the popular version control tool GitLab:

 CrateDB <small>container</small> An open-source distributed SQL database	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> database
 Elasticsearch <small>container</small> Open-source search and analytics engine	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> database
 Gitlab CE <small>container</small> Open-source end-to-end software development platform	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> development, project-management
 Minio <small>container</small> A distributed object storage server built for cloud applications and devops	<input checked="" type="checkbox"/> Update <input type="button" value="Delete"/> storage

Figure 11.34: GitLab Template in Portainer interface

Once the template has been selected, we must assign a network, volumes, and the ports that we will expose. In our case, port 80 is redirected to 9080, 443 to 9443, and 22 to 9022 .

In the following screenshot we can see the Port mapping assignation in Portainer interface:

Port mapping

Portainer will automatically assign a port if you leave the host port empty.

host	9080	→	container	80	TCP	UDP	<input type="button" value="Delete"/>
host	9443	→	container	443	TCP	UDP	<input type="button" value="Delete"/>
host	9022	→	container	22	TCP	UDP	<input type="button" value="Delete"/>

Figure 11.35: Port mapping in Portainer interface

Once deployed, the container related to GitLab is visible in the Containers list section:

The screenshot shows the Portainer interface with the following details:

- Containers** tab selected.
- Actions** bar: Start, Stop, Kill, Restart, Pause, Resume, Remove, Add container.
- Search** input field.
- Table Headers:** State, Quick actions, Stack, Image, Created, IP Address, Published Ports, Ownership.
- Rows:**
 - gitlab/gitlab-ce:latest (healthy, running, stack -, image gitlab/gitlab-ce:latest, created 2019-11-13 18:11:19, IP 172.17.0.2, ports 9022:22, 9443:443, 9080:80, ownership administrators).
 - portainer/agent:latest (running, stack test, image portainer/agent:latest, created 2019-11-13 18:06:54, IP 10.0.0.10, port 9001:9001, ownership administrators).

Figure 11.36: Container list in Portainer interface

In the following link, you can find the full documentation for what you could see its full potential <https://portainer.readthedocs.io/en/stable/index.html>

You can try a demo of the tool with the URL <http://demo.portainer.io> (log in with the username admin and password tryportainer).

Deploying Portainer to Docker Swarm Cluster

Portainer is compatible with the Docker engine and Docker Swarm. Katacoda provides a scenario for deploying Portainer to Docker Swarm Cluster in the URL <https://www.katacoda.com/portainer>. This scenario deploys Portainer to a Swarm cluster. The first step is creating a Docker Swarm Cluster with the command `docker swarm init`.

You can view the status of the Swarm Cluster with the `docker node ls` command:

The screenshot shows the Katacoda deployment interface for Portainer to Docker Swarm Cluster, Step 1 of 5:

Deploying Portainer to Docker Swarm Cluster

Step 1 - Deploy Swarm Cluster

This scenario deploys Portainer to a Swarm cluster. A two-node cluster has been created for you. If you would like more information on how this was created, complete the Katacoda scenario on [Getting Started With Swarm Mode](#).

You can view the status of the Swarm cluster with the command `docker node ls`

Terminal Output:

```
$ docker node ls
Error response from daemon: This node is not a swarm manager. Use "docker swarm init" or "docker swarm join" to connect this node to swarm and try again.
$ docker swarm init
Swarm initialized: current node (o1jqayqqnycub5ewrvf0oymb) is now a manager.

To add a worker to this swarm, run the following command:

  docker swarm join --token SWMTKN-1-2z8m2oz1d1ky3e84s78r6idqgw19yc3qqd27t8h30dy0vd92ub-3mc1fnw6ld
pltbkzlzemraku 172.17.0.77:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

$ docker node ls
ID          ENGINE VERSION      HOSTNAME      STATUS      AVAILABILITY      MANAGER ST
ATUS        VERSION
o1jqayqqnycub5ewrvf0oymb *   host01       Ready       Active           Leader
18.09.7
$
```

CONTINUE button.

Figure 11.37: Container list in Portainer interface

With the cluster configured, the next step is to deploy Portainer. Portainer is deployed as a container running on a Docker Swarm cluster or a Docker host.

To complete this scenario, deploy Portainer as a Docker Service. By deploying as a Docker Service, Swarm will ensure that the service is always running on a manager, even if the host goes down.

The service exposes the port 9000 and stores the internal Portainer data in the directory /host/data. When Portainer starts, it connects using the docker.sock file to the Docker Swarm Manager.

```
$ docker service create \
```

```
--name portainer \
--publish 9000:9000 \
--constraint 'node.role == manager' \
--mount type=bind,src=/host/data,dst=/data \
--mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
portainer/portainer \
-H unix:///var/run/docker.sock
```

When executing the previous command, you can see with the docker ps command that Portainer container is executing on port 9000 .

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
98046356e7ee       portainer/portainer:latest   "/portainer -H unix:..."
23 seconds ago      Up 19seconds       9000/tcp           portain-
er.1.e47c60moo81kge4k3ds1mj7ae
```

With Portainer running, it's now possible to access the dashboard and manage the cluster via a UI. The dashboard is running on port 9000 and can be accessed in the Portainer UI tab.

When entering the password, we access the user interface where we see how we have the Portainer container in running state:

The screenshot shows the Portainer Container list interface. At the top right, there are links for 'Portainer support', 'admin', 'my account', and 'log out'. Below the header, there are buttons for 'Containers', 'Columns', and 'Settings'. A toolbar at the top has buttons for 'Start' (green), 'Stop' (red), 'Kill' (orange), 'Restart' (blue), 'Pause' (grey), 'Resume' (yellow), 'Remove' (red), and '+ Add container' (blue). A search bar is present. The main table lists containers with columns: 'Name', 'State' (with a dropdown menu showing 'running'), 'Quick actions', 'Stack', 'Image', and 'Create'. One row is visible for the 'portainer' container, which is currently running. At the bottom, there are pagination controls for 'Items per page' (set to 10) and navigation arrows.

Name	State	Quick actions	Stack	Image	Create
portainer.1.9rl1kq9j3oknva9q5...	running	Stop Restart Logs File	-	portainer/portainer:latest	2019-11-

Figure 11.38: Portainer Container in execution

The next step is deploying annginx server using the Application templates list :

The screenshot shows the Portainer UI interface. At the top, there's a header with the title "Deploying Portainer to Docker Swarm Cluster" and a progress bar indicating "Step 4 of 5". Below this, a section titled "Step 4 - Deploy Template" is shown. A sub-section titled "Task - Deploy nginx" is highlighted. On the left, a sidebar shows a preview of the "nginx-web" template configuration. The main area is titled "Application templates list" and lists three templates: "Registry" (Docker image registry), "Nginx" (High performance web server), and "Httpd" (Open-source HTTP server). Each template has "Update" and "Delete" buttons.

Figure 11.39: Deploying nginx application template

The next step is selecting the nginx template, enter a name for the container, for example, nginx-web, and in Port mapping section we need mapping port 80 to the host port 80 .

The screenshot shows the configuration page for the "nginx-web" template. It includes sections for "Configuration", "Access control", and "Port mapping".

- Configuration:**
 - Name: nginx-web
 - Network: bridge
- Access control:**
 - Enable access control:
 - Administrators**: I want to restrict the management of this resource to administrators only
 - Restricted**: I want to restrict the management of this resource to a set of users and/or teams
- Port mapping:** [+ map additional port](#)

Portainer will automatically assign a port if you leave the host port empty.

host	80	→	container	80	TCP	UDP	
------	----	---	-----------	----	-----	-----	--

Figure 11.40: Deploying nginx application template

In the Container list , we can see that an instance of nginx has been deployed. Using the dashboard, you will see the state and be able to control the cluster.

In the following screenshot, we could get container details where we can see information related to Image, port configuration, environment variables, and labels:

Container details									
Image	nginx:latest@sha256:540a289bab6cb1bf880086a9b803cf0c4cefe38cbb5cdefa199b696145 25199f								
Port configuration	0.0.0.0:80 → 80/tcp								
CMD	nginx -g daemon off;								
ENTRYPOINT	null								
ENV	<table border="1"> <tr> <td>PATH</td><td>/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin</td></tr> <tr> <td>NGINX_VERSION</td><td>1.17.5</td></tr> <tr> <td>NJS_VERSION</td><td>0.3.6</td></tr> <tr> <td>PKG_RELEASE</td><td>1~buster</td></tr> </table>	PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin	NGINX_VERSION	1.17.5	NJS_VERSION	0.3.6	PKG_RELEASE	1~buster
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin								
NGINX_VERSION	1.17.5								
NJS_VERSION	0.3.6								
PKG_RELEASE	1~buster								
Labels	<table border="1"> <tr> <td>maintainer</td><td>NGINX Docker Maintainers <docker-maint@nginx.com></td></tr> </table>	maintainer	NGINX Docker Maintainers <docker-maint@nginx.com>						
maintainer	NGINX Docker Maintainers <docker-maint@nginx.com>								

Figure 11.41: Deploying ngnix application template

Docker Swarm administration with Portainer

Portainer provides us with a web GUI from which to manage a Docker Swarm Cluster. For example, you could configure your environment with 4 nodes, 2 of them master, and 2 other workers. These 4 nodes are configured to serve requests from my containers.

With the docker node ls command, we can get the list of nodes configured in the Docker swarm cluster.

```
$ docker node ls
```

In the following screenshot we can see the output of the previous command:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
olw40ii82gslxgfjtmy5h2ncr *	docker-master1	Ready	Active	Leader
1vvjz6c4bfe4ez16kzkyuc31o	docker-master2	Ready	Active	Reachable
q8em4kaftthkm1phkuuab48eau	docker-worker1	Ready	Active	
hb4ylcxtchc6kevimap7wxr9e	docker-worker2	Ready	Active	

Figure 11.42: Active node workers with Docker Swarm

With docker service command, you can view services have been deployed:

```
$ docker service ls
```

In the following screenshot we can see the output of the previous command:

ID	NAME	MODE	REPLICAS	IMAGE
tpmlzwsun4xt	httpd_httpd	global	4/4	httpd:2.4
9040dr4bjqtf	jboss_jboss	global	4/4	jboss/wildfly:latest
rhjjxlrqimj7	portainer_agent	global	4/4	portainer/agent:latest
rs5wrgflhmax	portainer_portainer	replicated	1/1	portainer/portainer:lates
oxj7bf6e36kj	tomcat_tomcat	global	4/4	tomcat:8.0

Figure 11.43: Active services with Docker

The Portainer installation can, therefore, be done by deploying it as another service within our cluster with the following YML file:

```
$ curl -L https://downloads.portainer.io/portainer-agent-stack.yml -o portainer-agent-stack.yml
```

Taking a look at the configuration file of the stack, we see that the agent is deployed in the global mode for the entire cluster, and the container containing the administration services is deployed in replication mode.

This is the content of the YML file:

```
version: '3.2'
```

```
services:
```

```
agent:
```

```
image: portainer/agent
```

```
volumes:
```

- /var/run/docker.sock:/var/run/docker.sock
- /var/lib/docker/volumes:/var/lib/docker/volumes

```
networks:
```

- agent_network

```
deploy:
```

```
mode: global
```

```
placement:
```

```
constraints: [node.platform.os == linux]
```

```
portainer:
```

```
image: portainer/portainer
```

```
command: -H tcp://tasks.agent:9001 --tlsskipverify
```

```
ports:
```

- "9000:9000"
- "8000:8000"

```
volumes:
```

- portainer_data:/data

```
networks:
```

- agent_network

```
deploy:
```

```
mode: replicated
```

```
replicas: 1
```

```
placement:
```

```
constraints: [node.role == manager]
```

```
networks:
```

```
agent_network:
```

driver: overlay

attachable: true

volumes:

portainer_data:

For deploying Portainer in the cluster using the previous file configuration, we can use the following command:

```
$ docker stack deploy --compose-file=portainer-agent-stack.yml portainer
```

Once the deployment is done, we can enter Portainer and see the services, as well as a status of the nodes that form the Docker Swarm Cluster:

Services					
<input type="checkbox"/>	Name	Stack	Image	Scheduling Mode	Published Ports
<input type="checkbox"/> httpd_httpd					
	httpd	httpd	httpd:2.4	global 4 / 4	<input checked="" type="checkbox"/> 80:80 <input checked="" type="checkbox"/> 443:443
	jboss_jboss	jboss	jboss/wildfly:latest	global 4 / 4	<input checked="" type="checkbox"/> 8080:8080 <input checked="" type="checkbox"/> 9990:9990
	portainer_agent	portainer	portainer/agent:latest	global 4 / 4	-
	portainer_portainer	portainer	portainer/portainer:latest	replicated 1 / 1 <input type="checkbox"/> Scale	<input checked="" type="checkbox"/> 9000:9000
	tomcat_tomcat	tomcat	tomcat:8.0	global 4 / 4	<input checked="" type="checkbox"/> 8180:8080

Figure 11.44: Active services with Docker

In this screenshot we can see the nodes that are part of the Docker Swarm Cluster:

Cluster status						
Nodes	4					
Docker API version	1.39					
Total CPU	4					
Total memory	6.27 GB					
Go to cluster visualizer						
Nodes						
<input type="checkbox"/> Search...						
Name	Role	CPU	Memory	Engine	IP Address	Status
docker-master1	manager	1	2.1 GB	18.09.1	192.168.1.131	ready
docker-master2	manager	1	2.1 GB	18.09.1	192.168.1.132	ready
docker-worker1	worker	1	1 GB	18.09.1	192.168.1.133	ready
docker-worker2	worker	1	1 GB	18.09.1	192.168.1.134	ready

Figure 11.45: Nodes that are part of the Docker swarm cluster

These tools allows you remotely or locally managing containers, start them, stop them, kill them, restart them, stop them paused, rename them, delete them, and add containers, create images, lift networks, monitor logs and containers in progress (CPU, memory, network use, processes, ...), run a console to access them, work with volumes and other interesting features.

Conclusion

In this chapter, we have reviewed Rancher and Portainer open source tools for managing your Docker containers, images, volumes, networks. These tools are compatible with the Docker engine and with other orchestration platforms like Docker Swarm and Kubernetes.

Once you enter the world of containers, there are many tools we can use in the Docker ecosystem, but I believe that Rancher and Portainer are the fastest, easiest, and best-supported options in the DevOps community.

Questions

Which tool consists of a single container that can be run on any Docker engine, and it can be implemented as a Linux container or a native Windows container?

Which platform has a Hosts section to visually manage the machines or instances of different clouds, either AWS (Amazon), Azure (Microsoft), and Digitalocean?

Which Rancher section allows you to deploy a container from the container dashboard and see the state of each one container from the Rancher interface?

Which parameter option you need to use in order to Portainer can manage the local Docker server with the docker run command?

Which check button we need to activate for showing templates related to container images like Docker registry or MySQL?