POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

DevSecOps pipelines improvement: new tools, false positive management, quality gates and rollback

Supervisors

Prof. Riccardo SISTO

Dr. Federico VIETTI

Candidate

Giovanni BERNARDO

28/10/2022

Abstract

DevSecOps, as extension of the DevOps paradigm, allows to integrate security inside applications and infrastructures from the beginning of the development, and to automate these security control activities. This development practice decreases the time necessary to make security checks, avoiding a ping-pong effect between developers and analysts, and allowing to save resources. A powerful DevSecOps instrument is the CI/CD pipeline: a sequence of steps that provides Continuous Integration (CI) and Continuous Delivery (CD), introducing automated security monitoring and providing a way to optimize the application development process.

The objective of this thesis is the improvement of already existent DevSecOps pipelines orchestrated by Jenkins, focusing: on the introduction of new tools, on the management of false positives and on the introduction of quality gates and rollback functionalities.
In this scenario cloud related technologies such as Docker and Kubernetes are used, with the purpose of hosting applications and tools.

After a brief introduction about DevSecOps, about pipelines and about the different kinds of analysis, the first part of the thesis analyses basic tools initially suggested by the company and then the newly discovered ones. There are tools for static analysis (SAST, SCA, Container security) based on a white-box approach, tools for dynamic analysis based on a black-box approach and a IAST tool that works in a grey-box mode.

The second part examines how false positives can be managed in DependencyCheck and ZAP. The choice of these two tools is given by the fact that they are already used in the company pipelines.
Nevertheless, a third solution using DefectDojo has been designed, in order to be more general and applicable also to other tools.

The third section describes how to implement a quality gate for ZAP and a quality gate suitable when a Vulnerability Management Tool, such as DefectDojo, is used. A quality gate can be seen as a security check to verify that the exposure of the software to vulnerabilities is under an identified threshold. If the threshold is exceeded than countermeasures such as the pipeline failure or a rollback operation have to be performed. The rollback functionality is the second topic treated in this section: it is a procedure used to go back to a previous version of a software/application, in this case applied when the software is considered

not secure, so if it does not pass the quality gate. This second part introduces a possible scenario where an application running on a Kubernetes cluster is classified as not secure through a quality gate, and for this reason it is necessary to perform a rollback: stop of the newer running version and re-start of the older and secure one.

In the last section, previously found and tested tools are evaluated and compared.
Each kind of instrument is compared separately: SAST tools are compared through a benchmark, Container security tools through the reports generated on public docker images, SCA tools through the reports generated on public projects, while DAST tools comparison is based on the analysis of vulnerable applications.
With regards to IAST tools, just one instrument of this type have been found and tested. Given the lack of competitors, a real comparison was thus not possible, but nevertheless an overview about how this tool works is submitted.
It is meaningful to say that no suitable benchmarks other than "OWASP Benchmark" have been found, and that this last point of the work takes into account, for the most part, only the total number of found vulnerabilities, without identifying the percentage of false positives.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Nowadays, DevOps methodology have been gaining popularity in the development field thanks to its capability to increase the cooperation between Development and Operations teams, leading to briefer development life cycles. Moreover, it allows to monitoring the performance, the infrastructure and the software distribution.
At the same time, companies has to face an increasing problem related to the security of the created applications: software is becoming day by day more complex, more features are available, new frameworks can be used and in general the technologies are evolving.
Due to the need of creating more robust software, a new approach is spreading: DevSecOps.
DevSecOps can be seen as an extension of the DevOps methodology, to which is included security in each stage of the software development life cycle. It is based on automation and allows to integrate security checks, in order to generate more secure products without delaying the needed production time.
A useful instrument in DevSecOps field is the CI/CD pipeline, which is used to automate a sequence of tasks and security checks.
Pipelines are composed by steps related to the building of images or related to code compilation, but also there are steps where security tools are used in order to perform security analysis.

An other important aspect is related to the usage of cloud technologies.
In these years the cloud is becoming more and more widespread thanks to its features such as scalability, elasticity and on-demand resource provisioning.
In order to follow the trend, recognizing its potentiality, in this thesis everything is carried out through Docker and Kubernetes, two of the main cloud technologies.

1.2 Thesis objective

Starting from a scenario where none of the following elements are already implemented, the purpose of this thesis is to improve basic DevSecOps pipelines through the introduction of new tools for static analysis, dynamic analysis, interactive analysis and through the introduction of the following three functionalities:

- False positive management in DependencyCheck, ZAP and DefectDojo.
- Quality gate for ZAP and with DefectDojo.
- Rollback of an application in a basic Kubernetes cluster.

The improvement of these pipelines is related to two aspects:

- The first one is related to the introduction of new tools. The idea is to look for tools that produce better results in less time, and that can be candidates to substitute the ones already used.
  Results are associated to the evaluation of these tools, evaluation based on metrics such as false positive rate, true positive rate, time needed to perform a scan and total number of findings. Better results means for example that the tool produces reports with a lower false positive rate but still with an high number of true positives.
- The second one is related to the introduction of functionalities that tries to make the security expert's job easier and more efficient.

Making the job of the security experts/developers easier, the entire development process become faster since the human efforts are substituted by automatic tasks that have only to be set up in the initial phase when the pipeline is built.
In particular:

- With regard to false positive management: this functionality is useful when security experts have to perform multiple scans on the same project.
  Without the false positive management, after a first scan when analysts verify that some of the findings are false positives, in the following runs the same findings will appear again, because they are not fixed since are not real vulnerabilities. So, analysts have to check many times the same false positive vulnerabilities, which leads to a reduced productivity.

- With regard to quality gate for ZAP and with DefectDojo: this functionality is useful to check the security exposure of the application.
  Without this functionality a full automation that takes in account security is not possible, because developers should scan the project/application, then manually verify the exposure and eventually publish the image/deploy the application.

- With regard to the rollback operation: this functionality is assumed to be applied with running applications. When a quality gate applied on a DAST tool fails, the new deployed version of the application is considered not safe and for this reason is necessary to come back to the previous version that has passed the quality gate, so to the not vulnerable one. Thanks to the rollback, is possible to automatically maintain in the developer environment the latest and most secure application version.
  Without it, as in the previous case, manual operations have to be performed, delaying the required time.

Notions about DevSecOps, pipelines and about the different types of analysis, useful to better understand the work, are reported in the chapter 2, while the quality gate and the rollback concepts are treated in the chapter 5.

1.3 Structure of the document

The following thesis is structured as follows:

- **Chapter 1**, *Introduction*: contains a brief introduction to the thesis.

- **Chapter 2**, *Theoretical foundations*: introduces and describes the fundamental elements that are part of this thesis, which means: DevSecOps, pipelines, and the different types of analysis (SAST, SCA, Container security, DAST, IAST).

- **Chapter 3**, *Development environment and tools*: describes the development environment with its related technologies (Docker and Kubernetes). Introduces the basic initial tools and the newly found ones.

- **Chapter 4**, *False positive management*: in this chapter are described the ways how false positives can be managed in DependencyCheck, in ZAP and through DefectDojo.

- **Chapter 5**, *Quality gate and rollback*: here are proposed two possible implementations of quality gate. A first implementation is applied when ZAP is

used and a second one when DefectDojo or in general another vulnerability management tool is available.
Furthermore, is described a possible way to introduce the rollback functionality in a pipeline while the application runs inside a Kubernetes cluster.

- **Chapter 6**, *Tool testing, evaluation and limits*: in this chapter is described the tool testing and evaluation phase, with its generated results.
  The testing phase is performed in different ways according to the available resources (such as benchmarks), and evaluation is performed through a comparison between the tools themselves. Tables and graphs are reported as conclusions foundations.

- **Chapter 7**, *Conclusions and future works*

Chapter 2

Theoretical foundations

In this chapter are introduced the main fundamental elements that compose the thesis, useful for the reader to better understand the work.
The first element introduced is the DevSecOps model, that can be considered an extension of the DevOps one, with its benefits and purposes.
Inside of DevOps we will find a useful way to improve the software development: the CI/CD pipelines, a chain of steps to optimize the software supply.
Beside the classical DevOps pipeline steps, with the security introduction is important to talk about the different kind of analysis that can be performed. In particular will be presented SAST, SCA, Container security, DAST, IAST, and under which assumption these techniques can be employed.

2.1 DevSecOps

DevSecOps can be seen as an extension of the DevOps software delivery model to which security practices are added. This practices involve the entire Software Development Life Cycle (SDLC), from its beginning.
Using this approach it is possible to find vulnerabilities before the product is completed and deployed in production, breaking down the cost and the time needed to patch these security flaws.

DevSecOps practises are considered important since digital transformation has become a fundamental requirement for almost all enterprises. This transformation includes: more software, cloud technologies and DevOps methodologies.

- **More software** means that more of the organization's risk becomes digital, making application security even more important than before.
- **Cloud usage** means that enterprises use new technologies that are easily accessible, redefining the concept of **security perimeter** [1].
  It also means that many of the IT and infrastructure risks are moved to the cloud, and others are becoming purely software defined, reducing many risks while highlighting the importance of permission and access management.
- Lastly, **DevOps** means a change to how software is developed and delivered, accelerating the cycle from the writing of the code to the delivery

Some of the benefits of the DevSecOps model include [2]:

- **Faster delivery**: In order to speed up the software delivery process, a possible solution is to include security checks inside the pipelines, allowing to find and to fix security flaws before the deployment.
- **Improved security**: In DevSecOps the security is introduced since the design phase: all the actors involved are responsible of the security aspects.
- **Reduced risks and costs**: Finding bugs and vulnerabilities before the deployment allows to reduce risks and operational costs.
- **Improving security integration and rate**: Time and cost of secure software delivery are reduced through the elimination of the need to retrofit security controls after the development.
- **Increase the value of the company**: A secure software and the usage of new technologies is able to increase the value of the software itself and the value of the company.

DevSecOps aims to build security into every stage of the delivery process and to establish a plan for security automation.
There are many security aspects and analysis that can be taken into account, for example:

[1]Security perimeter: a physical or logical boundary that is defined for a system, domain, or enclave; within which a particular security policy or security architecture is applied. - CNSS [1]

- **Static analysis, linters, and policy engines**: they can run any time a developer makes some changes in code. Used to verify if the custom written code contains vulnerabilities.
- **Software composition analysis**: it can be applied to check if open source third-party components have compatible licenses and are free of vulnerabilities.
- **Security integration checks / dynamic analysis**: code runs in an isolated sandbox in order to perform automated testing related to, for example, network calls, authorization and input validation.
  After this initial test phase, the code is deployed to a wider sandbox where further security integration tests are performed. Here it is possible for instance to test logging and access controls.

Thanks to the automated patching and configuration management, once the application reaches the production we can be almost sure it reaches an environment that is running the latest and/or most secure version of the software.

Nowadays, DevSecOps is associated to some combination of **Continuous Integration** and **Continuous Deployment (/Continuous Delivery)** systems in the form of a CI/CD **pipeline**.
Continuous Integration and Continuous Delivery principles assure that, during the development process, code correctness and code quality are checked.
Security measures can be adopted also inside a CI/CD pipeline: every time a developer builds the code, a CI/CD pipeline is executed, which means that a specific sequence of operations is performed, such as executing custom code or dependency related analysis, pushing code toward a remote repository, etc...

2.2 Pipelines

A CI/CD pipeline automates software delivery process within DevOps and DevSecOps models.
CI/CD pipelines build code, run tests (CI) and safely deploy a new version of the application (CD).
Continuous Integration (CI) is a software development practice in which all developers build, test and merge code changes in a central repository multiple times a day.
Continuous Delivery (CD) adds automation to the entire software release process.

With Continuous Integration, each change in code triggers an automated build test sequence for the given project, providing feedback to the developers. Through this process the code safeness is checked.
Continuous Delivery includes infrastructure provisioning and deployment. If also deployment is automated, then we talk about **Continuous Deployment** instead of Continuous Delivery.

A CI/CD pipeline is composed by many set of activities that a developer needs to perform to delivery a new version of a software product.
Each set of activities is considered as a **stage** (or phase) of the pipeline.

So, this kind of pipeline automates processes that are traditionally manual, with some benefits such as:

1. Reduction of human errors.
2. Speeding up and simplification of the developers' work.
3. Simplification of rolling back to a previous build version.

Pipelines can be evaluated through different parameters, for example:

- **Speed**: there are many factors related to speed, such as the time needed to complete the pipeline or the time needed to set a new one.
- **Reliability**: a reliable pipeline produces always the same output for a given input and doesn't have intermittent failures.
- **Accuracy**: the entire software delivery process is performed accurately, without leaving out any manual step.

A pipeline can terminate with failure, with a success or can be marked as unstable. The ending state is based on what happens during the execution. Typically a pipeline ends with a failure if one of its stages fails.

Here is represented a comparison between stages in DevOps pipelines and in DevSecOps pipelines:

Stage	**DevOps**	**DevSecOps**
Code Stage	code is written.	deploy and use linting tools and git controls to write secure code and to safely use passwords and API keys.
Build stage	code is combined with dependencies to build a runnable instance of the product. Programming languages such as Java are compiled, in some other cases images are built.	use SAST tools to track down flaws in code before deploying it on production. Usually these tools are specific to programming languages.
Test stage	developers find out whether their code works according to customer requirements through automated tests. In this phase code correctness and product behaviors are checked.	additionally to what happens with DevOps, here DAST tools are used to detect errors related to SQL injection, APIs, user authentication and authorization.
Release stage	the code is pushed toward a remote repository. In some cases this can be a preliminary step to activate the entire pipeline.	security analysis tools are used to perform vulnerability scanning and penetration testing. Since these tools need an high expertise, much time and so are costly, they should be used just before releasing the application.
Deploy stage	runnable instances are deployed in a specific environment.	as in DevOps

Table 2.1: DevOps stages vs DevSecOps stages

Information here reported are taken in part from: "CI/CD Pipeline: A Gentle Introduction." [3] and "DevSecOps Pipeline - A Complete Overview | 2022." [4]

2.3 Types of analysis: SAST, SCA, Container Security, DAST and IAST

2.3.1 SAST

Static Application Security Testing is a white-box [2] vulnerability scanning technique focused on assembly code, bytecode or source code.

Often developers don't have the security background to be able to avoid insecure programming patterns and know how to use secure APIs: here SAST comes into play as a part of application security.
SAST tools can run in a pipeline or even inside an IDE while writing the code, depending on the tool itself.
Many SAST tools identify the exact vulnerability position, highlighting the code; moreover these tools are able to provide tips and useful information about how to resolve the security flaws, making the fix workflow potentially easier.
On the other side, SAST tools are language-dependent and often have difficult to analyze code that cannot be compiled, such as Python, PHP, JavaScript.
SAST techniques can be applied to each phase of the software development life cycle.

There are many kind of code analysis and each of them is focused on a specific type of findings:

1. **Configuration analysis**
   - Checks the application configuration files
   - Ensures that the configuration is aligned with policies and security practices.
2. **Semantic analysis**
   - Code is split in token, syntax is examined and types are resolved.
   - This technique allows to find for example **SQL injections** or **buffer overflows**.

[2]White box testing: is a methodology for software testing in which the internal structure/design/implementation is known to the tester.

3. **Dataflow analysis**
    - Tracks the data flow across the application in order to determine if an input is validated before using.
    - Determines whether data coming from not trusted source is cleaned before consumption. Examples of not trusted source are: files, network, user input.
4. **Control flow analysis**
    - Checks the order of the program operations to detect potentially dangerous sequences, such as secure cookie transmission failure, uninitiated variables or misconfigurations of utilities before using.
5. **Structural analysis**
    - Examines language-specific code structures looking for not secure practices and techniques.
    - Identifies weaknesses in class design, declaration and use of variables and functions.
    - Identifies issues with generation of cryptographic material and hardcoded passwords.

SAST tools can be chosen taking into account different criteria, such as the following ones [5]:

- **Programming language**: SAST tools support just a subset of programming languages. This means that not all the SAST tools will work in each possible case.
- **Covered vulnerabilities**: a SAST tool is able to cover just a subset of vulnerabilities.
- **Custom rules**: tools provide a way to add custom rules.
- **Accuracy**: ability to avoid false positives, or in general to provide results near to the reality exposure state.
- **Compatibility**: a SAST tool should be able to work with other CI and development tools.

2.3.2 SCA

SCA is the acronym of Software Composition Analysis and it is a methodology used to analyze third-party components.
This analysis is performed to evaluate security, license compliance, and code quality.
Nowadays, most applications use third-party open source components and they may contain vulnerabilities that can be used as base for attacks. Actually SCA can be considered as a key pillar of application security programs.
A developer may directly include some open source packages in his code but those packages may rely on others open source packages that the developer did may not know about.
These in-direct dependencies can go several layers deep, leading the manual analysis to be long and tricky.

SCA tools may inspect package managers, manifest files, source code, binary files, container images, and more.
Identified components are listed in a Bill Of Material (BOM): an inventory of all assets of the project.
The Bill of Materials is then compared against vulnerability databases such as NVD or CVE, which have information related to known and common vulnerabilities.
Also other types of database can be consulted in order to discover licenses associated with the code and to analyse overall code quality.

Summing up, some of the benefits of SCA are [6]:

- Keep track of open source components automatically, avoiding to perform manual operations over hundreds of them.
- Detect weak points thanks to continuous monitoring: SCA tools constantly monitors and alerts when vulnerabilities are found.
- Vulnerabilities are identified and dealt automatically: advanced SCA provide also prioritized automated management tools to fix them up throughout the pipeline of SDLC.
- Risk management of licenses: SCA tools help to avoid the risk of a license violation by the organization.

2.3.3 Container security

In the last years containers usage is exponentially increasing. Given the high number of possible available images to choose from, securing containers has become a compulsory task.

VMWare, one of the leader companies in computing virtualization, defines container security as: "the process of using security tools and policies to protect all aspects of containerized applications from potential risks. Container Security manages risks throughout the environment, including all aspects of the software supply chain or CI/CD pipeline, infrastructure, and container runtime and life cycle management applications running on containers." [7]

Talking about Container security there are many layers where security can be applied, such as:

- The **container image** and **software inside**: the container image includes all the components that can be executed in the application. If there are vulnerabilities in the container image, the exposure increases during the production.
- **Host operating system**
- **Other containers** on the host and the interaction between containers.
- Container **networking** and **storage** repositories.
- The **runtime environment**, e.g. a Kubernetes cluster: there may exist newly discovered application vulnerabilities in old images or configuration changes may be occurred.

Each of these levels is important, but the only one that will be treated in this work is the first one which is related to the images and their software.
There are different ways to create a secure container image, following are proposed some possible tips to keep an high security level:

- **Secure custom code and its dependencies**: Containers expands the concept of "application code" moving towards a new scenario where applications are wrapped in a new environment, but the code of the image is still under the control of the developers.
  Assuming that developers have access to the source code, in this phase they can use SAST tools to analyze the custom code and SCA tools to analyze dependencies and libraries.

- **Build up with minimal base image from a trusted source**:
  This tip can be put in practice for example whether a Dockerfile [3] is used.
  The base image choice is one of the most important considerations when it comes to security. Smaller and minimal are the initial images, less will be the possible dependencies/code and therefore less will be the potential number of contained vulnerabilities.

- **Manage the tools and packages added to images throughout the development lifecycle**:
  Taking in account that Docker images are composed by many layers, the idea is to have in the first layer the application code and in the last layer the base image. Between this 2 external layers there should be middle layers with tools and framework used during the development.
  It is important to remove these middle layers before the publication of the image, because they are not necessary when the end of development is reached but they may still contains vulnerabilities.

[3]A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession [8]

2.3.4 DAST

Dynamic Application Security Testing (DAST) is a black-box [4] security testing methodology where an application is tested while running, and therefore this technique cannot be applied to an early development stage.
With the DAST approach, the application is analyzed in an "outside in" way, which means that the application status and its responses to simulated attack are examined. During the "attack", a tester provides inputs, and observes the outputs generated by the system under test. Responses to these simulations help to determine where the application is vulnerable and if real attacks can be performed.

DAST solutions can detect and help to protect against web application vulnerabilities, such as the ones described in the OWASP Top 10 [5]. Common flaws include SQL injection, cross site scripting (XSS), external XML entities (XXE), and cross-site request forgery (CSRF).
Moreover, DAST tools are able to find configuration and authentication errors and they can be used also to find flaws that are visible only when a users is logged in. For instance, the tool may try to run scripts or try to provide inputs when a dialog window is found: the purpose is to understand how the software handles the errors and if there is a way to exploit these flaws.

Before starting the real scan, the DAST tools crawl the web application, allowing the scanner to find all exposed inputs on pages within the web application, which are then tested.

SAST and DAST are both useful approach that should be both used during the development of an application; in the table below some differences between these two techniques are shown:

[4]Black box testing: is a software testing methodology where the functionalities of an application are examined without having information about its internal structures or workings. Here the tester acts as an attacker.

[5]OWASP TOP 10: is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

SAST	**DAST**
Find more vulnerabilities	Find less false positives
White-box approach	Black-box approach
"Inside-out" testing	"Outside-in" testing
Needs source or binary code	Needs a running application
Can be applied during the entire SDLC	Can be applied only at the end of the SDLC
Exhaustive analysis	Works on a limited input set
Cannot find environmental related nor running related vulnerabilities	Time consuming
Typically supports each software type (web application, web services ...)	It is language and technology agnostic

Table 2.2: SAST vs DAST

Is important to note the even if DAST and penetration testing may look similar, they are not the same: DAST offers systematic testing focused on the application in a running state, whereas penetration testing uses common hacking techniques and attempts to exploit vulnerabilities beyond the application, including firewalls, ports, routers, and servers, so with a more system-oriented approach.

2.3.5 IAST

Interactive Application Security Testing is a testing methodology where, similarly to DAST, a running application is tested in order to find vulnerabilities.
IAST methodology is based on the idea of combining SAST and DAST, in a way that may face newly complex applications.
The core of a IAST tool is an additional sensor (or agent) included in the analysed application code. These sensors keep track of application behaviour while the interactive test is running. With IAST only triggered code lines are analysed, finding less false positives if compared with SAST and on the other side finding more vulnerabilities than DAST, with also better performance with regard to the time needed.
Examples of such vulnerabilities could be hardcoded API keys in cleartext, not sanitizing users inputs, or using connections without SSL encryption.
Agent needs all the execution details but it's not necessary the access to the entire codebase: agent has to access at least to the triggered code, which means the elements that are used during the functional tests.
Since it analyses the triggered code, the agent has to be up during the whole application execution, in order to provide updated information about application status and to provide information about newly found vulnerabilities.
Moreover this approach is considered scalable, which means that can handle large applications without undue strain.

The downside is that IAST approach is tightly related to specific languages, frameworks and technologies; nowadays IAST tools do not support all the frameworks and languages, and also for this reason they are not yet widely adopted.

Information reported in this section are taken from: [5] [9] [10]

Chapter 3

Development environment and tools

This chapter is dived in 2 sections:

In the first one are described the technologies and instruments used to build the environment and to deploy analysis tools and applications to be tested: here are introduced **Docker**, **Kubernetes** and **Minikube**, a pre-built cluster used to save time since creating a custom Kubernetes cluster is not about this thesis.

In the second part are treated the security analysis tools used and tested inside the pipelines, focusing on their features.
This section is dived in two subsections: one for the basic tools initially suggested by the company and one for the additional and newly discovered ones.

3.1 Development environment

3.1.1 Docker

Docker is an open platform that allows to develop, to ship and to run applications. Docker gives the possibility to separate the applications from the infrastructures so the software delivery can be quicker: the time needed to run the code in production can be substantially reduced.

Docker is based on **images**: an image is a read-only template containing instructions to create a Docker container. Frequently, an image is based on another image, with some additional customization.
In order to build a custom image, a Dockerfile is needed. The Dockerfile is composed by a sequence of steps used to create the image and to run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have been changed are rebuilt, and this is what makes images so lightweight, small, and fast.
With Docker technology, software can be packed and then executed in a relatively well isolated environment called **container**. A container is lightweight running instance of an image and contains everything needed to run the application without relying on what is installed on the host, allowing to share the container itself among different machines. Thanks to the isolation and security properties, it's possible to run many containers on a given host at the same time. The isolation level of a container with respect to other containers and to host machine can be controlled by manipulating network, storage and other underlying subsystems.
A container can be started, stopped, moved or deleted, and can be attached to one or more networks and to one or more storages.
Moreover, a container is characterized by its image and by the configuration options that can be provided when the container itself is created or started.

Docker containers have some specific properties, such as:

1. **Standardization**: a container is represented in a standard way that allows it to be executed everywhere Docker is available.

2. **Lightweight**: containers shares the kernel of the host machine and this is why it is not necessary to have a OS for each application. This feature allows to have more efficient servers and to reduce licences costs.

3. **Security**: by default docker provides an high isolation level which protect the applications that run inside the containers.

4. **Resource isolation**: predictable application performance.

In the image below is proposed a simple description of the Docker architecture:

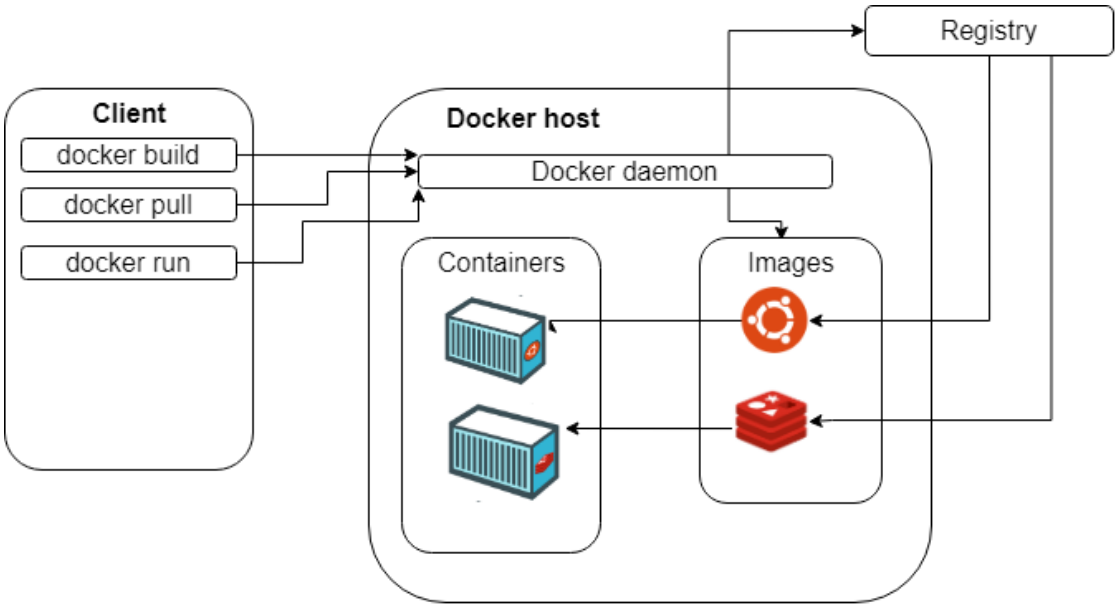

Figure 3.1: Docker infrastructure

The Docker client interacts with the Docker daemon, which carries the commands out: the daemon builds, runs, and distributes Docker containers. The Docker client can communicate with many daemons.
The Docker daemon and client can run on the same system, or can be connected remotely. They communicate using a REST API, over UNIX sockets or a network interface.

3.1.2 Kubernetes

Kubernetes is an open source, extensible and portable platform for managing containerized workloads and services, able to make easier both configuration and automation.
Kubernetes is based on containers, which are a good solution to distribute and run applications. In a production environment it is necessary to manage the containers and to ensure that there will be no interruptions; for instance, if a container goes down, another one has to replace it. Kubernetes provides a framework to run distributed systems resiliently, taking care of scaling and failover for applications, and providing also deployment patterns.
Among all the features, Kubernetes provides:

- **Load balancing**: A Kubernetes container can be reachable from the outside of the cluster by using a DNS name or its IP address. Moreover, Kubernetes allows to redirect the traffic to different containers if it is too high for a single one. In this way is possible to have more stable services.

- **Self-healing**: To keep available services, Kubernetes manipulates the containers, restarting the ones that fail, replacing them and killing the ones that do not match the user-defined health checks.

- **Storage orchestration**: Kubernetes allows to automatically mount a chosen storage system, which can be for example a local storages or a public cloud one.

- **Automated rollbacks**: In Kubernetes is possible to describe the desired state for the deployed containers. Kubernetes is able to change the actual state trying to reach the desired one. For example, Kubernetes can be automated to create new containers for the deployment, remove existing containers and adopt all their resources to the new container.

Once that some of the fundamental features are described, it is possible to go a bit more deeper and see which is the infrastructure under Kubernetes.
A Kubernetes cluster is composed by a set of nodes where containerised applications run. Every cluster has at least one worker node that hosts some Pods.
The control plane role is to handle nodes and Pods in the cluster, making global decisions (such as the scheduling) and responding to specific events (such as starting a new pod for a ReplicaSet). In order to provide an high availability and fault tolerance, the control plane should run across different computers while the cluster should run across different nodes.

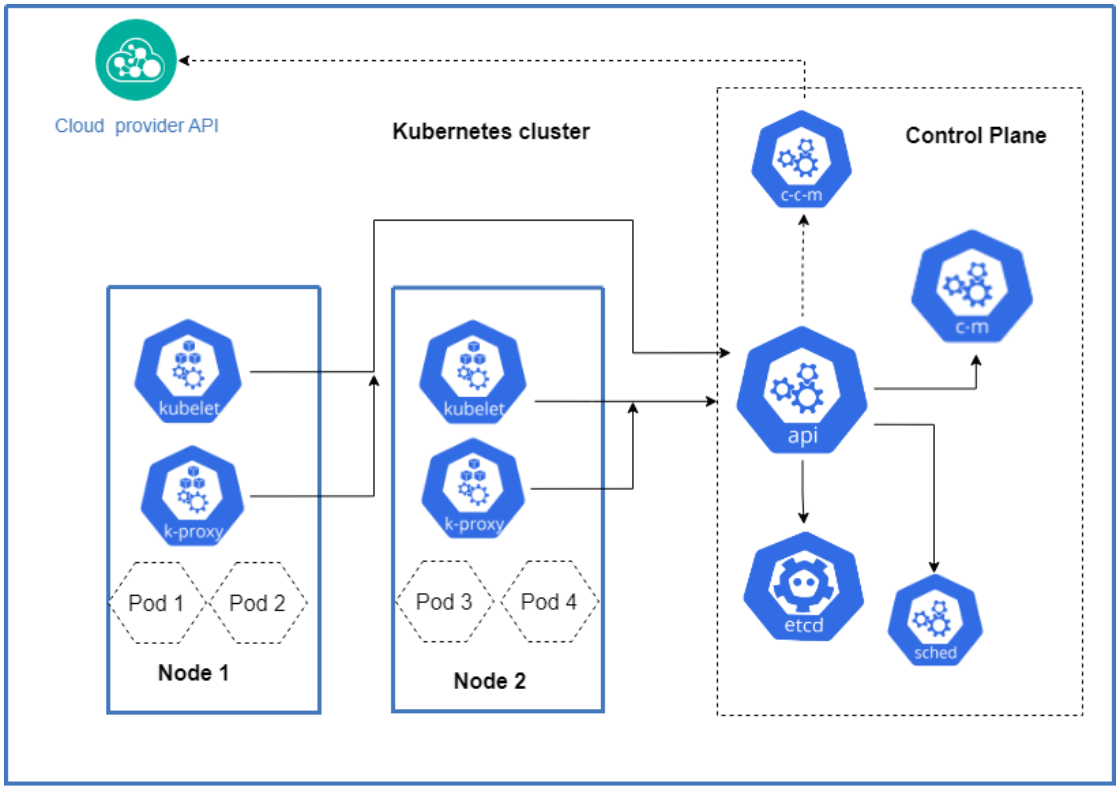

Figure 3.2: Components of Kubernetes

Kubernetes provides many different resource types useful to customize the cluster specifying properties and to deploy applications; the most common Kubernetes resources are:

- **Deployment**: A Deployment provides declarative updates for Pods and ReplicaSets. The deployment controller changes the actual state to the desired and described one. Deployments are used to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments

- **Pod**: Pods are the smallest deployable units of computing that can be created and managed in Kubernetes.
  A Pod wraps one or more containers, with shared storage and network resources. Each Pod has a specification for how to run the containers.

- **ReplicaSet**: The ReplicaSet's purpose is to maintain always stable the number of replica Pods which are running: it is usually used to guarantee the availability property.

- **Service**: Exposes an application running in the cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

Information reported are taken from Kubernetes official website [11]

3.1.3 Local cluster configuration

After a brief overview about Docker and Kubernetes, this subsection is dedicated to the choice of a ready Kubernetes cluster, where tools and application will be deployed. There exist many different possible clusters, such as Minikube, Kind, K3s.
Some of their fundamental features are:

- **Multi-node/multi-cluster** support
- **Dashboard** availability
- **Supported architectures** (AMD64, ARMv7...)
- **S**upported container runtimes (Docker, CRI-O, gvisor...)
- First start **time**/next starts time
- **Memory** requirements
- **Root access**: if root privileges are required.

It is important to take into account that these three named clusters, such as all the other existent ones, are evolving and are updated with a certain rate with performance improvement and feature addition.
After researches and comparisons, the chosen cluster was Minikube. Minikube provides a useful dashboard to interact with the cluster, an easy installation, all the basic K8S features and requires only 2GB of RAM.
Once Minikube setup is completed, is possible to deploy the needed tools, such as **Jenkins**, **Sonarqube** and **DefectDojo**.
To perform the deployment, different paths can be followed. For example is possible to use Helm [1], as done for DefectDojo, or it is possible to use custom yaml files to create Kubernetes resources needed to describe application (resources like "Deployment" and "Service"), as done for Jenkins and Sonarqube.

[1]Helm: a package manager to install and manage Kubernetes applications. It allows to find, share, and use software built for Kubernetes.

With respect to the deployment of tools through yaml files, only official docker images have been used.
When a tool is deployed, it has to be configured to communicate with other actors inside the cluster. In this case tools has to interact with Jenkins, which orchestrates the pipelines.
While some tools interact with Jenkins passing through Kubernetes, others use plugins (such as Snyk for SCA and DependencyCheck) or are installed directly inside the Jenkins pods and are used through the Command Line Interface; this last technique is applied by tools such as Snyk (Code analysis and Container Security analysis), CodeQL, Trivy and Grype.

3.2 Initial tools

In this section is proposed a brief overview about the initial tools proposed by the company.
The purpose is to give a basic knowledge about these tools, allowing to understand their role in a pipeline and in this thesis.

3.2.1 Jenkins

Jenkins is a self-contained and open source server, used to automate different kind of tasks such as those related to building, testing, delivering and deploying of software, as well as those concerning security.
Jenkins is the main and fundamental tool on which the whole process is based. It allows the writing and the execution of the CI/CD pipelines. Moreover it is highly extensible through the usage of plugins that allow to easily interact with other tools and to integrate new Dev(Sec)Ops stages.
A downside of this approach is that even the plugins may contain vulnerabilities or their development may be interrupted. For this reason is important to perform periodic checks also regarding this aspect.

Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed. In this specific scenario, as previously said, Jenkins is executed thanks to Deployment and Services resources in Kubernetes.

Below is proposed a graphical representation of a pipeline in Jenkins, where all the steps are executed in the correct way and the pipeline ends with success.

Figure 3.3: Jenkins pipeline example

3.2.2 Sonarqube

Sonarqube is a tool used to perform automatic code analysis.
Sonarqube can be integrated with Jenkins, and allows to define customs quality gates for each project. Moreover, it can be connected through the so called "webhook" to other tools, such as DefectDojo.
It is no necessary to specify the programming languages adopted to write the application because Sonarqube is able to recognize them and applies for each language a different set of rules, named "profile". It is possible to create different custom profiles characterized by a different set of rules; each profile can be applied to more projects or can be used as the default one for that specific language.
Whenever one of these rules is not respected, an **issue** is generated. Sonarqube issues can be classified in 3 categories:

- **Bug**: a problem that may leads to errors or to unexpected behaviours during the execution.
- **Vulnerability**: a part of the code that could be attacked.
- **Code smell**: a problem related to the maintainability, that makes the code unclear and difficult to maintain.

Similarly to vulnerabilities, Sonarqube identifies the so called "**Security hotspots**". They differ from the vulnerabilities because they need a manual control before to decide if a fix is required or not.
Even if security hotspots are found it is possible that the general application security may not be affected. On the other side, when a vulnerability is found, developers are in a scenario where the flaw has an impact on the security, and a quick fix has to be applied.

Sonarqube performs source code static analysis for all the supported languages but for some of them it is available also the analysis of the compiled files (such as .class file for java, .dll file for C).

Information here reported are taken from SonarQube official website [12]

3.2.3 DependencyCheck

Dependency-Check is a Software Composition Analysis (SCA) tool that attempts to find publicly disclosed vulnerabilities within a project's dependencies.
Dependency-check can be used in many ways: it provides a Jenkins plugin, a CLI (command line interface), an Ant task and a Maven plugin.
The Jenkins plugin is the way DependencyCheck is used in this thesis. It allows to

perform analysis and to have a quick sight of the analysis result with generated metric, trends and findings. Here, it's possible to configure also a threshold in order to fail the build or putting it into a warning/unstable state.
The core engine contains some analyzers that inspect the project dependencies and collect information about these dependencies; information collected are referred to as evidence. The evidences are used to recognise the CPEs [2] for each single dependency. When a CPE is identified, a list of entries associated to the CVEs (Common Vulnerability and Exposure) [3] is shown in the report. For specific technologies other third-party services and data sources are used (such as OSS Index or RetireJS).

Anyway it is important to remember that even if a CVE exists, this does not mean the risk applies to each possible scenario.

Dependency-check analysis are based on the National Vulnerability Database (NVD) data, hosted by NIST.

3.2.4 ZAP

ZAP is an open source tool specifically designed to test web applications and it is both flexible and extensible.

ZAP can be seen basically as a "man-in-the-middle proxy", that intercepts and inspects messages exchanged between tester's browser and web application, modifying the contents if needed, and then forwarding each packet to the destination.
The presence of another proxy in the same network is not a problem: in that case ZAP can be connected to it, after a proper configuration.
ZAP can be used as a daemon process, as a Docker container or as a standalone application provided with a GUI (Graphic User Interface).
Inside the CI/CD pipelines ZAP is used as a Docker container, because it allows to launch automated scans without a direct human interaction.
ZAP is constantly evolving and new features are frequently added. Actually a new framework for the automation is under development, which is based on the usage of .yaml files. The idea behind this Automation Framework is to provides

[2]NIST: "CPE is a structured naming scheme for information technology systems, software, and packages. CPE includes a formal name format, a method for checking names against a system, and a description format for binding text and tests to a name. [13]"

[3]RedHat: "CVE, short for Common Vulnerabilities and Exposures, is a list of publicly disclosed computer security flaws. When someone refers to a CVE, they mean a security flaw that's been assigned a CVE ID number." [14]

high flexibility and the possibility to customize ZAP works without loosing the automation [15].
ZAP let developers to perform high customized scans, but the basic two are:

- Passive: the crawling of the web application is performed and pages are analyzed in order to find possible way to interact with the application from the outside (such as through textboxes, scripts ...).
- Active: once pages are analyzed, ZAP perform an active scan trying to attack the web application through the identified weak elements.

ZAP provides 2 ways to perform the crawling of web application:

- ZAP spider: looks for URLs by examining HTML code inside the answers sent by the web application. This kind of spider is fast but it is not always efficient when working with web applications based on AJAX that generates resources using JavaScript.
- ZAP AJAX: explores web application exploiting the browser and following generated URLs.
  This kind of spider is slower than the previous one and need an additional configuration.

3.2.5 DefectDojo

DefectDojo is an open source security orchestration and vulnerability management platform, that allows to manage application security program, to maintain product and application information, triage vulnerabilities and to push findings to systems like JIRA and Slack. DefectDojo in its core is a bug tracer, that provides also a way to trace issues among multiple projects and test cycles, allowing fine-grained reporting.

With DefectDojo is possible to reduce the time needed to log found vulnerabilities. This happens thanks to a system based on templates that allows to describe vulnerabilities, thanks to the import of reports generated by vulnerability scanners and thanks to metrics; moreover DefectDojo supports both manual activities and automatic ones based on CI/CD pipelines.

DefectDojo uses a hierarchical way to represent the vulnerabilities. The main elements of this structure are:

- **Product type**: allows to distinguish different "class" of products. For this purpose any logic attribute can be used, such as the development team or the business unit id.

- **Product**: this is the name of the tested item. It can be for example a project or a specific product.

- **Engagements**: allow to aggregate tests according to the time they are performed. Engagements have a name, a status, a time line and some others attributes. There are two types of engagement: Interactive and CI/CD.
  An interactive engagement is typically an engagement conducted by an engineer, where findings are usually uploaded by the engineer.
  A CI/CD engagement, as it's name suggests, is for automated integration with a CI/CD pipeline.

- **Tests**: are a group of activities conducted to discover flaws in a product. Tests have a start and end date and are defined by a test type.

- **Findings**: represent a flaw discovered while testing. They can be categorized with five severity levels: Critical, High, Medium, Low, and Informational.

- **Endpoints**: represent testable systems defined by their IP address or Fully Qualified Domain Name.

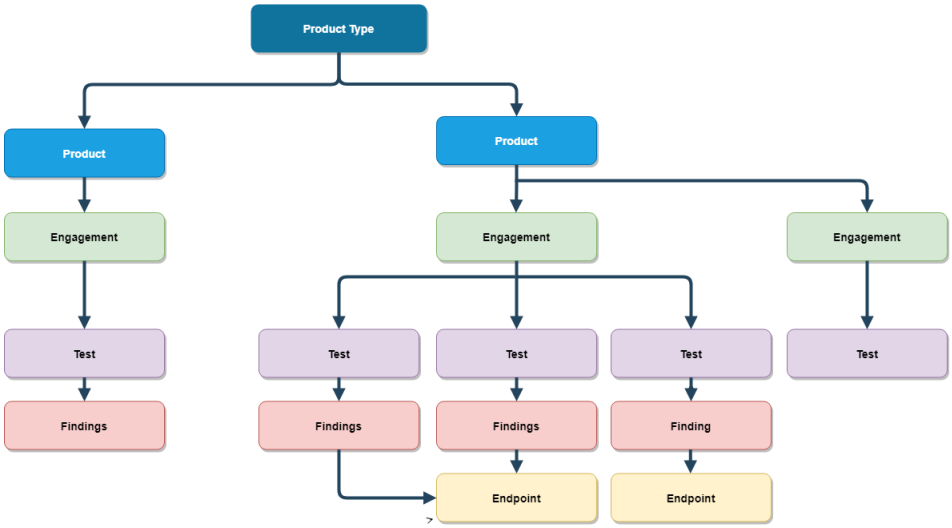

Figure 3.4: DefectDojo hierarchy representation

This hierarchy allows to obtain a good flexibility and can be adapted to different organizational and security structures.

3.3 Additional tools

In this section the additional found tools are introduced.
These tools can be divided in those that are related to static analysis, those that are related to dynamic analysis and at the end there is the only one which is based on the IAST approach.
As in the previous section, the purpose is to give a basic knowledge about these tools, allowing to understand their role in a pipeline and in this thesis.

The research of these new tools was performed preferring the Open Source software, or at least looking for a free version of them. An other relevant, but not essential, aspect was the direct connection between Jenkins and the tool, for example performed through a plugin for Jenkins: anyway many tools provide a CLI and produce reports which are stored where the tool itself runs, and so a plugin is not necessary.
Working with DefectDojo, the possibility to import the generated reports there, was a plus.
The only tested tool not reported here is ESLint, because even if it can be installed and used through a CLI, in my opinion it produces results that are more useful during the coding, so when used inside an IDE.

Static Analysis tools:

- **SAST**:
  Snyk
  CodeQL
- **SCA**:
  Snyk
- **Container security**:
  Trivy
  Grype

Dynamic Analysis tools:

- StackHawk
- Arachni

IAST tool: Contrast Community Edition

3.3.1 Snyk

Snyk is a commercial tool that proposes also a free plan, which is limited in features and in maximum number of scans.
Snyk can perform different types of analysis:

- Static Application Security Testing (SAST) - Snyk Code
- Software Composition Analysis (SCA) - Snyk OpenSource
- Container security - Snyk Container
- Infrastructure security - Snyk Infrastructure as Code
- Cloud security - Snyk Cloud

In this thesis will be treated only the functionalities related to SAST, SCA and Container security.
As for all the SAST tools, Snyk supports only a limited set of languages; among them there are Java, .NET, PHP, Ruby, Scala.

Differently from other tools which exploit the National Vulnerability Database (NVD) [16] to perform the analysis, Snyk uses a proprietary database maintained by a dedicated research team. This database, called Snyk Intel Vulnerability Database, combines public resources, elements coming from the developers community, proprietary researches and Machine Learning techniques.

In order to interact with Jenkins, a plugin can be used. Unfortunately it does not satisfy the need to use all the functionalities previously identified: through the plugin is available only the dependencies analysis (SCA), while to perform the static analysis of the code and to perform container security analysis is necessary a different approach, which is based on the usage of the Command Line Interface (CLI). An alternative to the CLI is the usage of APIs, but it is available only with Business and Enterprise plans, which are not free.

The installation of Snyk inside the Jenkins container, in order to perform SAST and Container security analysis, needs NodeJS, which has to be installed before Snyk.

3.3.2 CodeQL

CodeQL is an SAST tool used to automate security checks and to perform variant analysis.
In CodeQL, code is treated like data: the different flaws such as bugs and security vulnerabilities are represented as **queries**. After the database generation (extracted from the codebase), queries can be executed against it. For this purpose standard queries are provided, but it is possible to create and use custom ones, written in a particular language called QL.

CodeQL exploits the variant analysis: it is a process that starts from a known vulnerability (considered as a seed) and tries to discover similar issues in the code. It is a technique used by security engineers to identify potential vulnerabilities, and ensure these threats are properly fixed, also across multiple codebases.

Once standard or custom queries are chosen, it is possible to iterate over them to find in an automatic way the variants of the initial issue.

CodeQL analysis can be considered as a sequence of three steps:

- Creation of a CodeQL database:
  During the creation of a database, each source file in the codebase is analysed, in order to extract a relational representation.
  For compiled languages, CodeQL monitors the build process: whenever a source file is compiled, CodeQL collects all the relevant information about it. This set of information includes **name binding and type** information (**semantic data**) and the, **abstract syntax tree** (**syntactic data**).
  Moving towards interpreted languages, the extractor works directly on the source code, resolving its dependencies.
  It is common that a codebase contains many different languages; in these cases CodeQL generates one database for each language, one at a time.
  The code is represented in a hierarchical way. In particular, it is represented also through the data flow graph, the control flow graph and through the abstract syntax tree.

- Analysis of the database:
  In this phase queries (standard and custom) run against the database to find vulnerabilities.

- Reading the results:
  Here the results show the found potential vulnerabilities.
  There are queries that show only a specific location in the code, while others

show a series of locations that highlights a path across the control flow or across the data flow graph.

3.3.3 Trivy

Trivy is a tool that includes many security scanners that look for different security issues, and different targets where it can find those issues.
Trivy targets are:

- Container Images
- Filesystems
- Git repositories
- Kubernetes clusters or resources

While Trivy scanners look for:

- OS packages and software dependencies in use (SBOM, Software Bill Of Materials)
- Known vulnerabilities (CVEs)
- "Infrastructure as Code" misconfigurations (for Kubernetes, Docker, Terraform...)
- Sensitive information and secrets

Trivy analysis are based on Aqua vulnerability database and many other sources.

Working with CI/CD pipelines, deployment and integration can be performed by installing the binary and specifying the target. Trivy uses a database that supports automatic updates without requiring database dependencies and middlewares.

3.3.4 Grype

Grype is an open source vulnerability scanner that provides a centralized service for the analysis of container images and filesystems, looking for known vulnerabilities. Successor of Anchore Engine, it is able to find vulnerabilities for major operating system packages:

- Alpine
- Amazon Linux
- BusyBox
- CentOS
- Debian
- Distroless
- Oracle Linux
- Red Hat (RHEL)
- Ubuntu

Moreover, it can find vulnerabilities for language-specific packages:

- Ruby (Gems)
- Java (JAR, WAR, EAR, JPI, HPI)
- JavaScript (NPM, Yarn)
- Python (Egg, Wheel, Poetry, requirements.txt/setup.py files)
- Dotnet (deps.json)
- Golang (go.mod)
- PHP (Composer)
- Rust (Cargo)

When a vulnerability is found, the version of the identified vulnerable element with the fixed vulnerability is proposed if available .

A downside is that Grype supports only some image formats, which are Docker, OCI and Singularity.

Trivy can be easily installed by downloading the binaries from the official Github repository, to call the executable whenever a scan has to be performed.

3.3.5 StackHawk

StackHawk is a commercial dynamic application and API security testing tool based on the open source project ZAP.
It has two parts:

- the HawkScan Scanner: can run anywhere (laptop, server, Kubernetes, or in a CI/CD pipeline).
- the StackHawk Platform: results are collected on the StackHawk Platform, where it is possible to analyze, communicate, and track findings to resolution.

HawkScan uses a YAML file to configure the scanner.
There are many configuration settings available to tune HawkScan, such as the authentication which is configurable for the specific tested application, or such as the usage of OpenAPI specification.

As for the most part of analysis tools, also in this case the findings are categorized by their risk severity, based on the OWASP Risk Rating Methodology. The risk levels with their description are:

- High: Findings with significant impact and likelihood of exploit, usually with a known corresponding CWE or CVE attached to the vulnerability.
- Medium: Findings with significant impact or ease of exploit.
- Low: Informational and low-impact discoveries, as well as security suggestions.

StackHawk provides for each vulnerability the affected routes, evidences, how to reproduce the test and how to fix the vulnerability. Through the platform dashboard (StackHawk website) it is possible to use the Validate action to generate a curl command with correct HTTP verb, headers and data fields to recreate the potential attack.

It is important to remember that during the whole analysis, StackHawk server has to be reachable. Offline scans are not allowed.

StackHawk can be executed through its Docker image, and this is the way it is used in this thesis.

3.3.6 Contrast Community Edition

Contrast is a commercial suite of tools, which provides also a free version (Contrast Community Edition), used to measure the security of an application.

Contrast Community Edition provides the following functionalities of the paid platform solutions, such as Contrast Assess, Contrast SCA and Contrast Protect.

- **Contrast SCA**: analyses vulnerabilities related to open source third-party components.
- **Contrast Assess**: it is the strong suit of this set of tools. Contrast Assess comes into play during the test phase implementing the IAST approach.
  Thank to IAST approach development teams can secure each line of code thanks to the continuous detection and prioritisation of vulnerabilities, guiding the team towards the risks elimination.
  Moreover, allows to check which are the used vulnerable third-party components, avoiding to flag as vulnerable unused dependencies.
  In order to apply IAST on the analyzed application, an agent must be included inside of it. There is not a unique way to use this tool on an application: different technologies (Java, Node, Kotlin...) needs a different deployment approach.
  The agent has to be authenticated and has to communicate with the Contrast Server during the whole execution.
- **Contrast Protect**: provides a protection that blocks attacks and reduces false positives, helping developer teams prioritize vulnerability backlogs

Contrast provides also a Plugin for Jenkins that can be used to perform a quality gate. Unfortunately it seems to be not available in the free version.
Others Contrast Community Edition's limitations from the paid platform are language support (Java, .NET Core), only one application can be onboarded at a time and also information about found vulnerabilities are are obfuscated. These limitations lead to the impossibility to use the free version in an enterprise scenario.

Chapter 4

False positives management

Thanks to the Application Security Testing (AST), developers and security experts have the possibility to find security flaws that an attacker could use to compromise the software. If a security tool is able to recognise the vulnerabilities, developers can fix them, increasing the application security level.
Application security tools produce results that can be grouped in two categories:

- **True Positive**: an existing vulnerability correctly identified to be so.
- **False Positive**: a vulnerability that does not exist but that is identified anyway in the codebase.

On the other side, there are other two categories of vulnerabilities that will not be reported:

- **True Negative**: a vulnerability that does not exist and that is correctly not identified in the codebase.
- **False Negative**: a vulnerability that exists in the codebase but that is not identified.

A false detection can occur when a cyber security tool detects, within a non-malicious object, a signature identical to that of a known unhealthy object. For example, a vulnerability scanner could identify a software plugin as potentially vulnerable even if its current version is patched.
A possible reason to the generation of false positives is that the vulnerability scanner may not have access to all the required information needed to have an higher precision.
False positives in application security have a negative impact on the work of cyber-security experts, developers, and of the entire business.

The massive amount of time lost in managing and analysing false positives (i.e. false alarms) results in time lost dealing with real alarms. Indeed, both software and human resources will be used for an alert that should not be raised.
When a tool generates many false positives, in addition to wasting time, confidence in that tool is impaired. False positives are costly errors and result in a reduced productivity.

An ideal tool should maximize the number of true positives and true negatives, while should bring to zero the false negatives and false positives.

Below is shown a graphical way proposed by OWASP to evaluate tools, based on false positives and true positives [17]. Computing the False Positive Rate and the True Positive Rate allows to place the tools on this graph, where each area assumes a different meaning as described in the graph itself:

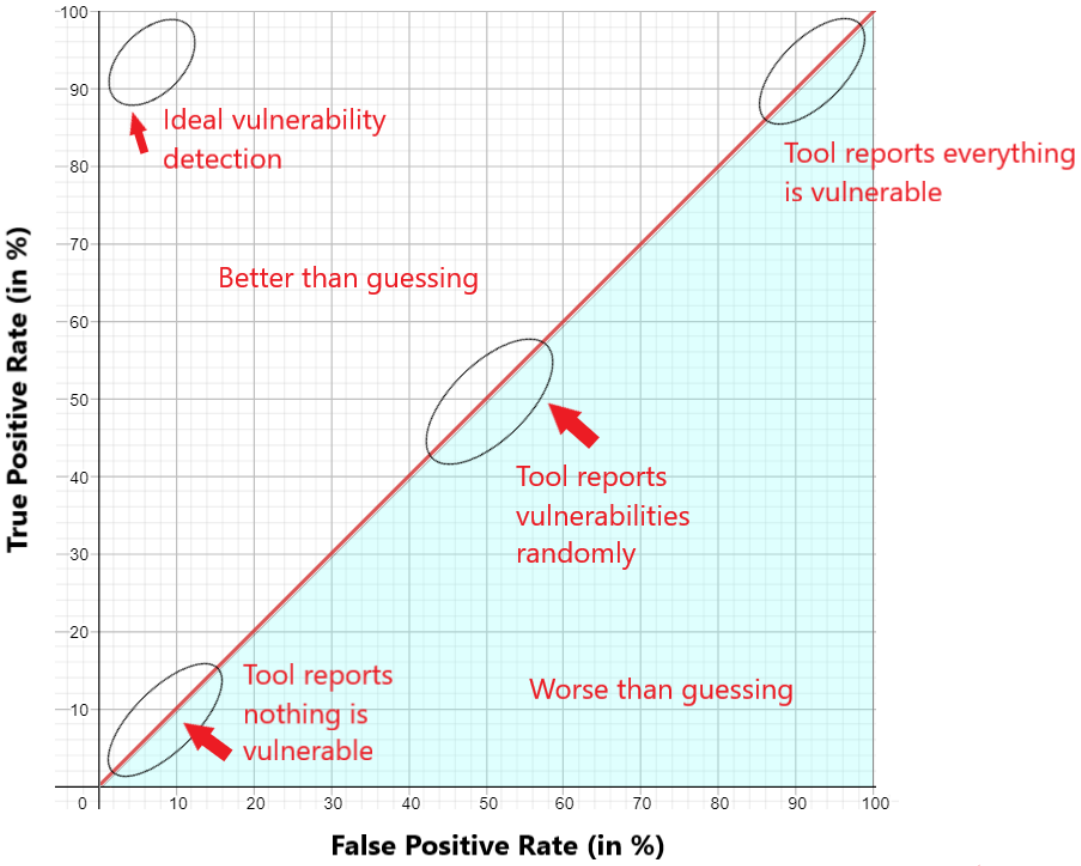

Figure 4.1: OWASP tool evaluation graph

Starting from an initial scenario where false positives were not handled, in the two following sections is described how to manage them in DependencyCheck and in ZAP.
The choice of DependencyCheck and ZAP is made by the company because these two tools were already used in existent CI/CD pipelines.
Nevertheless, a third solution using DefectDojo has been designed, in order to be more general and applicable also to other tools.

4.1 Dependency Check

Talking about DependencyCheck, in order to exclude some identified false positives we have to use an .xml file with a specific structure that can be found in the project documentation.
Once the basic components of the file structure are written, it is possible to specify each finding that has to be suppressed in the next analysis.
There are different ways to identify a false positive inside the .xml file and for this reason we have to keep in mind that there are many fundamental fields that can be used, if combined, to specify one or more elements to be suppressed. Some of these fields are CPE, CVE [1], SHA1, SHA256 [2] and filepath (that can be also a regular expression).
This means that there are several things that can be suppressed: individual CPEs, individual CVEs, all CVE entries below a specified CVSS [3] score and more. The most common approach is suppressing CPEs based on SHA hashes or filepath.
Regarding to CVEs we have to consider that flaws that impact more than one product get separate CVEs. For shared libraries, standards or protocols, the flaw gets a single CVE only if there is no way to use the shared code without being vulnerable. Otherwise each affected codebase or product gets a unique CVE. [14]
For this reason it is important to suppress only specific CVEs associated to specific elements, because it is possible to find the same CVE in different part of the code: in some part it may be a real vulnerability while in others it may be just a false positive. If we suppress findings working only with CVEs or in general using rules that implies a wide application range such as "all the entries below a specific CVSS

[1]CVE, standing for Common Vulnerabilities and Exposures, is a list of publicly disclosed computer security flaws.

[2]SHA: The Secure Hash Algorithms are a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST)

[3]CVSS: The Common Vulnerability Scoring System (CVSS) is an open framework for communicating the characteristics and severity of software vulnerabilities.

score", it may be possible to create false negatives.

Since the high number of possible combinations of the previously named fields, in the snippets below are reported some examples that show how to suppress a subset of findings in DependencyCheck working with these parameters.

dc_fp.xml

```
<suppress>
    <notes><![CDATA[
    This suppresses a CVE identified by OSS Index using the
    vulnerability name and packageUrl.
    ]]></notes>
    <packageUrl regex="true">^pkg:maven/org\.eclipse\.jetty/jetty-
    server@.*$</packageUrl>
    <vulnerabilityName>CVE-2017-7656</vulnerabilityName>
</suppress>
<suppress>
    <notes><![CDATA[
    This suppresses cpe:/a:csv:csv:1.0 for some.jar in the "c:\path\
    to" directory.
    ]]></notes>
    <filePath>c:\path\to\some.jar</filePath>
    <cpe>cpe:/a:csv:csv:1.0</cpe>
</suppress>
<suppress>
    <notes><![CDATA[
    This suppresses any jboss:jboss cpe for any test.jar in any
    directory.
    ]]></notes>
    <filePath regex="true">.*\btest\.jar</filePath>
    <cpe>cpe:/a:jboss:jboss</cpe>
</suppress>
<suppress>
    <notes><![CDATA[
    This suppresses a specific cve for any test.jar in any directory.
    ]]></notes>
    <filePath regex="true">.*\btest\.jar</filePath>
    <cve>CVE-2013-1337</cve>
</suppress>
```

dc_fp.xml

```
<suppress>
    <notes><![CDATA[
```

```
    This suppresses a specific cve for any dependency in any
    directory that has the specified sha1 checksum.
    ]]></notes>
    <sha1>384FAA82E193D4E4B0546059CA09572654BC3970</sha1>
    <cve>CVE-2013-1337</cve>
</suppress>
<suppress>
    <notes><![CDATA[
    This suppresses all CVE entries that have a score below CVSS 7.
    ]]></notes>
    <cvssBelow>7</cvssBelow>
</suppress>
<suppress>
    <notes><![CDATA[
    This suppresses false positives identified on spring security.
    ]]></notes>
    <gav regex="true">org\.springframework\.security:spring.*</gav>
    <cpe>cpe:/a:vmware:springsource_spring_framework</cpe>
    <cpe>cpe:/a:springsource:spring_framework</cpe>
    <cpe>cpe:/a:mod_security:mod_security</cpe>
</suppress>
<suppress>
    <notes><![CDATA[
    This suppresses false positives identified on spring security.
    ]]></notes>
    <gav regex="true">org\.springframework\.security:spring.*</gav>
    <vulnerabilityName regex="true"></vulnerabilityName>
</suppress>
<suppress until="2020-01-01Z">
    <notes><![CDATA[
    This suppresses a specific cve for any dependency in any
    directory that has the specified sha1 checksum. If current date is
     not yet on or beyond 1 Jan 2020.
    ]]></notes>
    <sha1>384FAA82E193D4E4B0546059CA09572654BC3970</sha1>
    <cve>CVE-2013-1337</cve>
</suppress>
<suppress until="2020-01-01Z">
        <notes><![CDATA[
        Suppresses a given CVE for a dependency with the given sha1
    until the current date is 1 Jan 2020 or beyond.
        ]]></notes>
        <sha1>384FAA82E193D4E4B0546059CA09572654BC3970</sha1>
        <cve>CVE-2013-1337</cve>
</suppress>
```

Source: [18]

It is meaningful to note that in some cases when a vulnerability is marked as false

positive inside the .xml file, maybe new vulnerabilities can be found in the next analysis: this is due on how DependencyCheck analysis is performed.

Here is reported a test to demonstrate how the suppression of false positives can be implemented and to explain how DependencyCheck performs its analysis.

1) Starting from the initial report of a first DependencyCheck analysis:

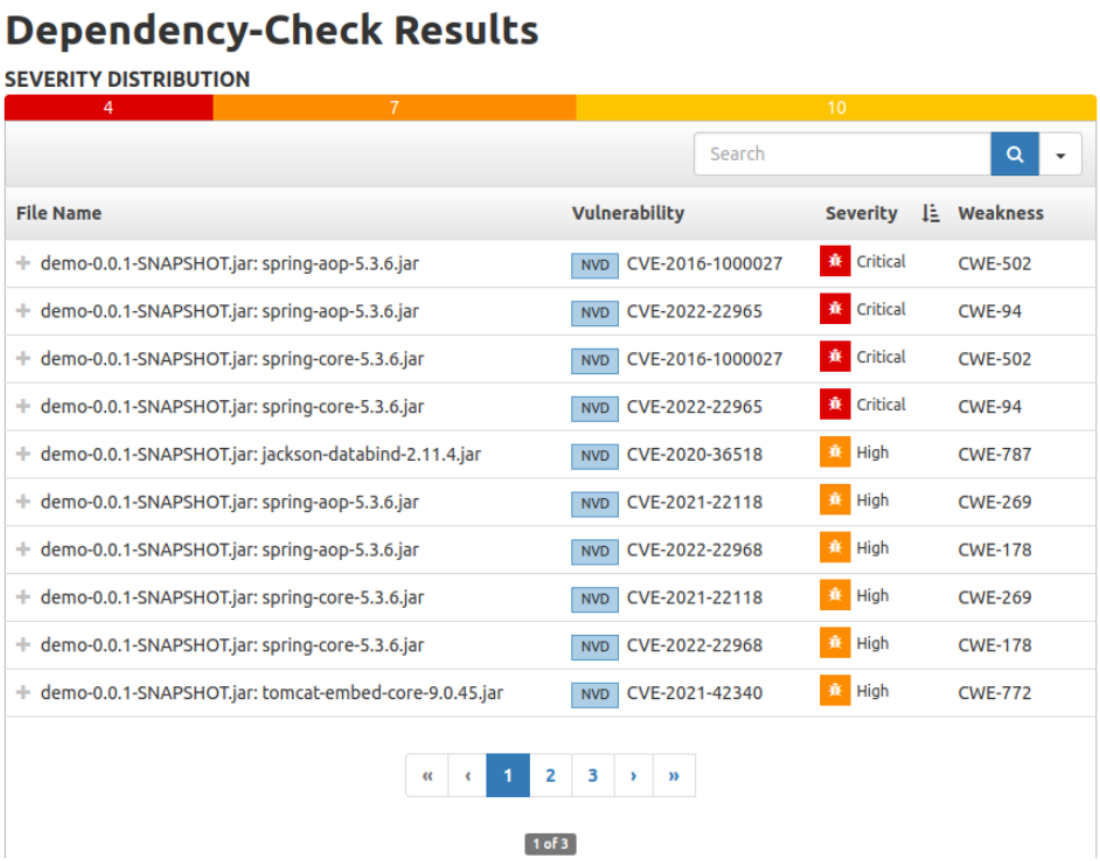

Figure 4.2: DependencyCheck first report

2) Create the xml file and put inside of it the items to be suppressed; in this case are introduced two vulnerabilities with the same CVE but with a different path and SHA checksum.
Additional useful information to customize suppression entries, such as the SHA checksum or filepath, can be found inside the DependencyCheck report.

```
<?xml version="1.0" encoding="UTF-8"?>
<suppressions xmlns=
"https://jeremylong.github.io/DependencyCheck/dependency-suppression.1.3.xsd">

   <suppress>
      <!-- This suppresses a specific cve for any dependency in any directory
      that has the specified sha1 checksum. -->
      <!-- spring-aop-5.3.6 -->
      <sha1>8f91f60f628075701fde72bb5a43a33feeb27e93</sha1>
      <cve>CVE-2022-22965</cve>
   </suppress>

   <suppress>
      <!-- This suppresses a specific cve for any dependency in any directory
      that has the specified sha1 checksum. -->
      <!-- spring-core-5.3.6 -->
      <sha1>f9290db7324194921c236ad9a940467f55304fa7</sha1>
      <cve>CVE-2022-22965</cve>
   </suppress>
</suppressions>
```

Figure 4.3: DependencyCheck false positives xml file

3) Repeat the analysis using the previous created .xml file with the items to be suppressed.

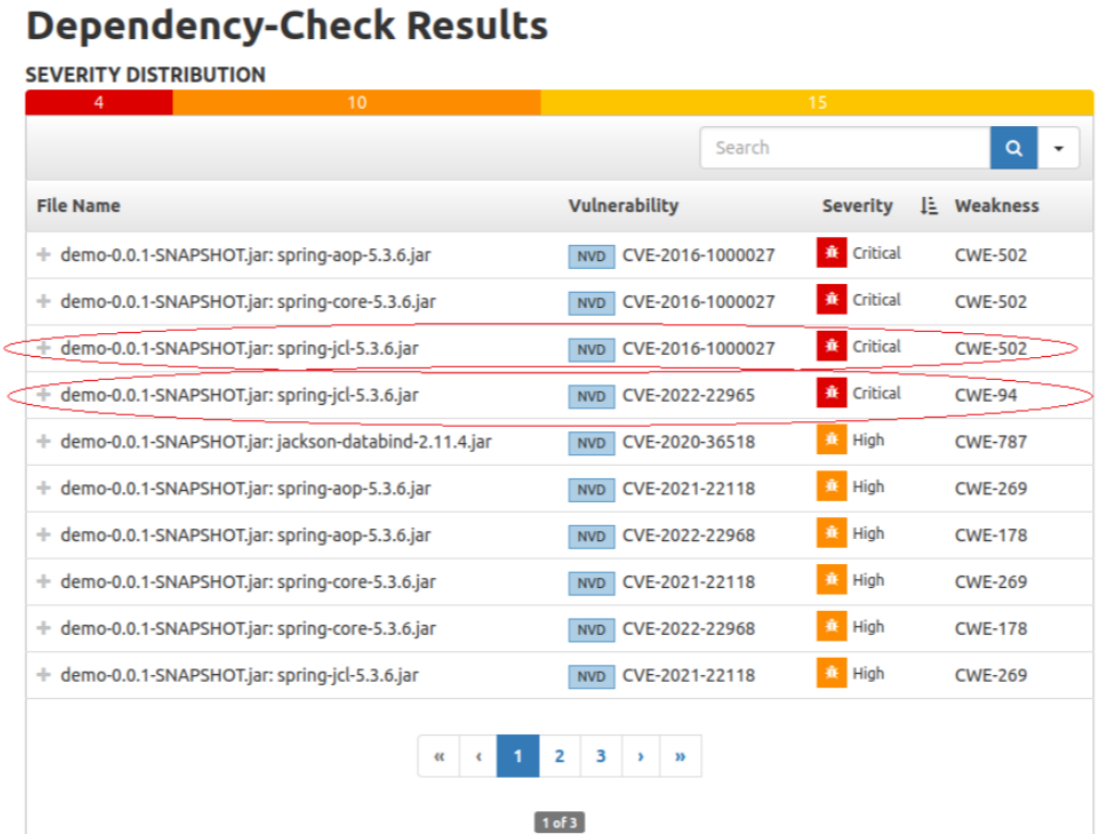

Figure 4.4: DependencyCheck report with FP management

As expected, the report shows that the two suppressed items are not inside the new report, but new vulnerabilities are discovered and the total amount of findings is increased: with the second analysis DependencyCheck finds 4 critical, 10 high and 15 medium risk level vulnerabilities.
Observing the dependencies tree in figure 4.5 we are able to note that in the image 4.4 two of the newly discovered and circled vulnerabilities are sub-dependencies of spring-core:5.3.6 (the file related to one of the items previously suppressed).
Also the others not shown findings follow the same principle: they are sub-dependencies of suppressed elements.

```
[INFO]    +- org.springframework:spring-core:jar:5.3.6:compile
[INFO]    |  \- org.springframework:spring-jcl:jar:5.3.6:compile
[INFO]    +- org.springframework:spring-test:jar:5.3.6:test
[INFO]    \- org.xmlunit:xmlunit-core:jar:2.7.0:test
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Figure 4.5: Fragment of a dependencies tree

The new findings are in a branch that is below the one containing the suppressed vulnerabilities.
This happens because DependencyCheck by default seems to stop its search through a branch when it finds a node with a vulnerability, ignoring all the ones that can exist below the found dependency.
This means that every time a finding is marked as false positive in DependencyCheck, developers or security experts have to be prepared to check also new vulnerabilities in the following DependencyCheck runs.

4.2 ZAP

ZAP false positive management is strictly related on which is the way ZAP is used. If ZAP is used through a GUI the false positive handling is carried out in a first way, while if ZAP is executed with a CLI based approach, the handling is performed in a second and different way.
First of all, we have to consider that we are using ZAP's docker images since they provide an easy way to automate ZAP, especially in a CI/CD environment.
I started from the following Jenkins stage where ZAP was executed as a Docker container, exploiting the "baseline" scan type :

```
def URL
pipeline{
    agent any
    environment{
        tmpDirectory="/var/comune"
        host="http://testhtml5.vulnweb.com/"
        ZAP_REPORT="reportTweety.xml"
    }
    stages{
        stage('ZAP analysis'){
            steps{
                sh(script: "docker run -v ${tmpDirectory}:/zap/wrk/:rw --network=host
                -t owasp/zap2docker-stable:s2022-03-02 zap-baseline.py
                -t ${host} -x ${ZAP_REPORT} -d -I", returnStdout: true)
                archiveArtifacts artifacts: "**/*.xml"
            }
        }
    }
}
```

Figure 4.6: ZAP initial pipeline

From this starting point I tried three different solutions, but just the last one is really working as expecting.

The **first** tried solution is the nearest to the starting point: it keeps using the standard baseline analysis, exploiting the possibility to re-write the scan rules.
ZAP gives the possibility to make changes to the scan rules through a configuration file passed to Docker when ZAP container is started.
In this file, each rule is identified by a RuleID and has an associated action that is performed if the rule matches.

The possible actions are:

- WARN: an alert is raised.
- IGNORE: the rule is ignored.
- FAIL: the scan fails if the rule matches.

```
10028   WARN    (Open Redirect)
10029   WARN    (Cookie Poisoning)
10030   WARN    (User Controllable Charset)
10031   WARN    (User Controllable HTML Element Attribuite (Potenzial XSS))
10032   WARN    (Viewstate)
10033   WARN    (Directory Browsing)
10034   WARN    (Heartbleed OpenSSL Vulnerability (Indicative))
10035   WARN    (Strict-Transport-Security Header)
10036   WARN    (HTTP Server Response Header)
10037   WARN    (Server Leaks Information via "X-Powered-By" HTTP Response Header Fields(s))
10038   IGNORE  (Content Security Policy (CSP) Header Not Set)
10039   WARN    (X-Backend-Server Header Information Leak)
10040   WARN    (Secure Pages Include Mixed Content)
```

Figure 4.7: ZAP rules example

order to increase the accuracy and to avoid that a rule matches more than one vulnerability, it is also possible to specify the URL to ignore by regex patterns. The added rules should have the following structure:
`<rule-id> OUTOFSCOPE <regex>`
e.g.:

```
# Ignore the specified URL for Autocomplete in browser
10012    OUTOFSCOPE    https://www.example.com/test.html
# Ignore all URLS containing '.js' for all scan rules
*    OUTOFSCOPE    .*\.js
```

Figure 4.8: ZAP out of scope rule example

Anyway, this approach has a problem: even if the output printed on the console is correct, which means that `OUTOFSCOPE` rules (our false positives) are correctly ignored in this phase, the same ignored findings are not excluded in the file-based report.
This is a consistency issue already reported and known by ZAP developer team,

but actually not resolved.
Due to this problem, this solution is valid only if the subsequent checks are based on what the standard output reports, otherwise, if checks are performed on generated file-based reports (such as an .xml file) this solutions is not valid.

The **second** tried solution requires a deeper understanding of how ZAP works.
ZAP provides a way to override some of its functionalities, through the so called "hooks".
Hooks allows to pass custom python scripts to ZAP when container is started, in the practice allowing to change the way ZAP works.
Following an article found on Medium [19], I tried to change the way ZAP alerts were raised.
The custom script aims to read the entire alert list and to filter it, by checking if each alert exists in a hand-crafted false positives list.
During the testing phase, even if the custom hook was correctly uploaded and executed, the report was still containing the false positive elements.

This possible solution is not easy to be implemented and requires that the ZAP functionalities does not change the way they work, but it would be possible to study it more in deep trying to make it work properly.

The **third** and last possible solution is the only one that works according to the company needs, which means that can be applied to generated file-based reports.
It uses a new and still under development framework, called `Automation Framework`.
The Automation Framework allows to configure the ZAP analysis through a .yaml file. Inside this yaml file, the analyst has to specify each step (called **job**) that needs to be performed.
Among the possible steps, there is the `alertFilter` one, that allows to override the standard rules, identified also in this case by a RuleID.
It is possible to set custom rules to be applied only to specify vulnerabilities, by exploiting some fields like: `urlRegex`, `attackRegex` and `evidenceRegex` .
With this approach is possible to generate a report where false positives are correctly marked to be so.

To conclude this section about false positives management in ZAP, a practical test of this feature is shown below.

1. Perform the first ZAP analysis based on a initial automation file:

```yaml
env:                                # The environment, mandatory
  contexts :                        # List of 1 or more contexts, mandatory
    - name: "test.context"          # Name to be used to refer to this context in other jobs, mandatory
      urls:  ["http://testhtml5.vulnweb.com"] # A mandatory list of top level urls,
                                    # everything under each url will be included
  vars:                             # List of 0 or more variables,
                                    #can be used in urls and selected other parameters
  parameters:
    failOnError: false              # If set exit on an error
    failOnWarning: false            # If set exit on a warning
    progressToStdout: true          # If set will write job progress to stdout
jobs:
  - type: spider                    # The traditional spider - fast but doesnt handle modern apps so well
    parameters:
      context: "test.context"       # String: Name of the context to spider, default: first context
      user:                         # String: An optional user to use for authentication, must be defined
                                    # in the env
      url: "http://testhtml5.vulnweb.com" # String: Url to start spidering from, default:
                                          # first context URL
      maxDuration:                  # Int: The max time in minutes the spider will be allowed to run for,
                                    # default: 0 unlimited
      maxDepth: 6                   # Int: The maximum tree depth to explore
      maxChildren:                  # Int: The maximum number of children to add to each node in the tree
  - type: activeScan                # The active scanner - this actively attacks the target so should only
                                    # be used with permission
    parameters:
      context: "test.context"       # String: Name of the context to attack, default: first context
      user:                         # String: An optional user to use for authentication, must be defined
                                    # in the env
      policy:                       # String: Name of the scan policy to be used, default: Default Policy
      maxRuleDurationInMins:        # Int: The max time in minutes any individual rule will be
                                    # allowed to run for, default: 0 unlimited
      maxScanDurationInMins:        # Int: The max time in minutes the active scanner will be
                                    # allowed to run for, default: 0 unlimited
  - type: report                    # Report generation
    parameters:
      template: "traditional-xml"   # String: The template id, default : modern
      theme:                        # String: The template theme, default: the first theme defined
                                    # for the template (if any)
      reportDir: "/zap/wrk"         # String: The directory into which the report will be written
      reportFile: "reportTestHtml5"  # String: The report file name pattern,
                                     #default: {{yyyy-MM-dd}}-ZAP-Report-[[site]]
      reportTitle: "Report title"   # String: The report title
      reportDescription: "Description" # String: The report description
```

Figure 4.9: ZAP Automation Framework initial configuration

In this file, excluding the report generation job, parameters are loosely customised: to be precise only the `context` [4] and the starting URL fields are personalized.
The `spider` job is used to automatically discover new resources (URLs) on a particular Site. It begins with a list of URLs to visit, called `seeds`. The

[4]Contexts are a way of relating a set of URLs together. Is expected that a context will correspond to a web application.

Spider then visits these URLs, identifying all the hyperlinks in the page and adding them to the list of URLs to visit; the process continues recursively as long as new resources are found.
The `activeScan` job attempts to find potential vulnerabilities by using known attacks against the selected targets.
These two jobs and parameters are the minimum elements needed to perform the requested analysis.

2. Identify the false positives and insert them inside the alertFilter section of the automation file.
   The alertFilter job will have a structure similar to the following one:

```yaml
- type: alertFilter                # Used to change the risk levels of alerts
  parameters:
  alertFilters:                      # A list of alertFilters to be applied
    - ruleId: 10017                  # Int: Mandatory alert rule id
      newRisk: "False Positive"      # String: Mandatory new risk level, one of 'False Positive', 'Info',
                                     # 'Low', 'Medium', 'High'
      context: "test.context"        # String: Optional context name, if empty then a global alert
                                     # filter will be created
      url:                           # String: Optional string to match against the alert,
                                     # supports environment vars
      urlRegex:                      # Boolean: Optional, if true then the url is a regex
      parameter:                     # String: Optional string to match against the alert parameter field
      parameterRegex:                # Boolean: Optional, if true then the parameter is a regex,
                                     # supports environment vars
      attack:                        # String: Optional string to match against the alert attack field
      attackRegex:                   # Boolean: Optional, if true then the attack is a regex
      evidence: 'script src="http://bxss.s3.amazonaws.com/ad.js></script>' # String: Optional string
                                     # to match against the alert evidence field
      evidenceRegex:                 # Boolean: Optional, if true then the evidence is a regex
```

Figure 4.10: ZAP Automation Framework AlertFilter Job

Starting from the configuration in figure 4.9 the job alertFilter is added and customized in this case with just one rule. This rule specifies that the findings associated to the ruleId 10017 and that match with the indicated evidence are marked as False Positive.
It is possible to add other rules following the same principle and exploiting the available alertFilter fields.
The spider and activeScan jobs generate a kind of first temporary report which is the input for the alertFilter. The alertFilter step aims to examine the pre-raised alerts, checking if any entry of the temporary report matches with any of its defined rules: for each match, the information about that entry are overwritten (e.g the risk level).

In essence: only the alertFilter job influences the management of false positives.

3. Perform a second ZAP analysis, based on the modified automation file obtained by the union of the job in figure 4.10 and of the file in figure 4.9

4. Check the new generated report where, as expected, just a specific instance of the rule 10017 is marked as false positive:

```
<alertitem>
    <pluginid>10017</pluginid>
    <alertRef>10017</alertRef>
    <alert>Cross-Domain JavaScript Source File Inclusion</alert>
    <name>Cross-Domain JavaScript Source File Inclusion</name>
    <riskcode>1</riskcode>
    <confidence>2</confidence>
    <riskdesc>Low (Medium)</riskdesc>
    <confidencedesc>Medium</confidencedesc>
    <desc><p>The page includes one or more script files from a third-party domain.</p></desc>
    <instances>
    </instances>
    <count>6</count>
    <solution><p>Ensure JavaScript source files are loaded from only trusted sources,
    and the sources can't be controlled by end users of the application.</p></solution>
    <otherinfo></otherinfo>
    <reference></reference>
    <cweid>829</cweid>
    <wascid>15</wascid>
    <sourceid>1</sourceid>
</alertitem>
<alertitem>
    <pluginid>10017</pluginid>
    <alertRef>10017</alertRef>
    <alert>Cross-Domain JavaScript Source File Inclusion</alert>
    <name>Cross-Domain JavaScript Source File Inclusion</name>
    <riskcode>1</riskcode>
    <confidence>0</confidence>
    <riskdesc>Low (False Positive)</riskdesc>
    <confidencedesc>False Positive</confidencedesc>
    <desc><p>The page includes one or more script files from a third-party domain.</p></desc>
```

Figure 4.11: ZAP analysis result with FP

4.3 Use of a Vulnerability Management Tool

In this section is treated how DefectDojo can be used to manage false positives, but a similar reasoning can be applied to each vulnerability management tool.

The usage of a vulnerability management tool seems to be a solution to overcome all the problems related on how each vulnerability scanner tool handles false positives.
When a report containing vulnerabilities is imported into DefectDojo, it is possible to mark those vulnerabilities as false positives through the Graphic User Interface or even through the use of APIs, providing to developers a unified way to tag vulnerabilities as false positive, as active, as duplicated and even more.
Below are reported the test phases of this functionality with a DependencyCheck report, but the same procedure can be adapted to any supported one:

1. The report is imported into DefectDojo from the Jenkins pipeline through the use of the APIs.

```
curl -k -X POST -q  -H "Authorization: ${AUTHORIZATION}"
-F "close_old_findings=true" -F "file=@${DependencyCheck_Report}"
-F "minimum_severity=Info" -F "engagement=${ENGAGEMENT_ID}"
-F "verified=false" -F "active=true"
-F "scan_type=Dependency Check Scan" -F "scan_date=${TargetEnd}"
    ${DEFECTDOJO_URL}/api/v2/import-scan/

```

2. Vulnerabilities are marked as false positives.
   In the image below I considered the finding related to the CVE-2020-15084 as a false positive.

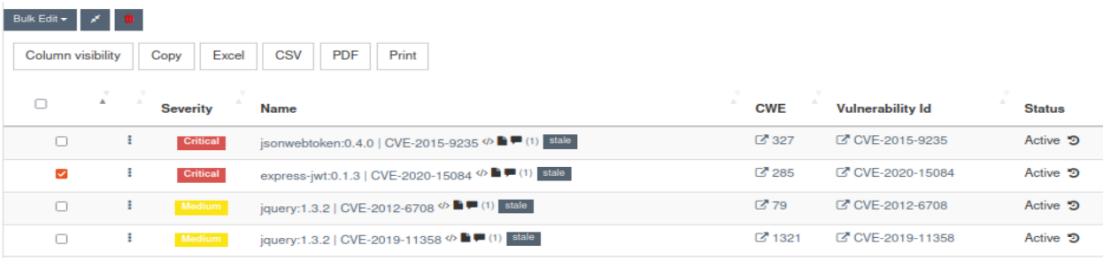

Figure 4.12: DefectDojo FP management 1

3. Through this window it is possible to change the status and to tag the selected vulnerabilities. Here the vulnerability is marked as `False Positive` and as `Mitigated`.

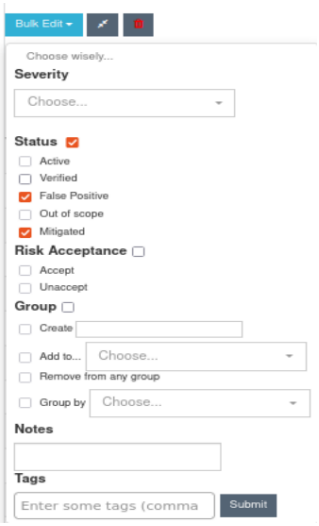

Figure 4.13: DefectDojo FP management 2

4. Once the changes are applied, the selected vulnerability has a new status according to the previous chosen one.

Figure 4.14: DefectDojo FP management 3

5. Through this window is possible to filter the findings to be shown, according to custom parameters. Here are reported the parameters used to exclude false positives and duplicates.

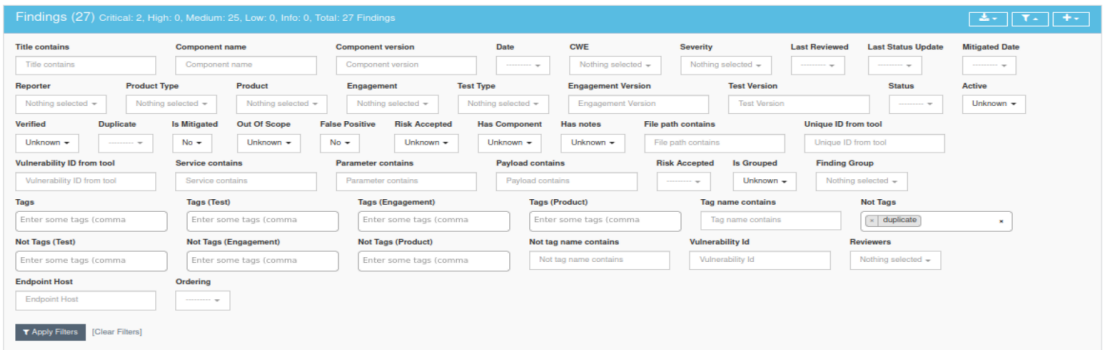

Figure 4.15: DefectDojo FP management 4

6. Below is presented the graphical result that shows only the filtered vulnerabilities. Note that the previously checked as false positive finding is not shown (CVE-2020-15084).

Figure 4.16: DefectDojo FP management 5

It is important to note that once a vulnerability is marked as false positive, even in the followings imports it will be marked as before, without requiring again a human action or an update through APIs.

Compared with the two previously presented approach, this solution is as already said more general but, on the other side, DefectDojo and in general a vulnerability management tool is a standalone tool which requires additional resources and efforts to be maintained: in a quite small environment with limited resources, or in a scenario where just a tool is used and the tracking of the vulnerabilities is not necessary, this approach might be not recommended.

A last consideration is that this approach seems to be very effective also when it is needed to work with quality gates: this case is proposed in the next chapter.

Chapter 5

Quality gates and rollback

5.1 Quality Gates

A quality gates can be seen as a security check based on an the analysis of reports generated by different tools.
Usually quality gates are employed to block a pipeline (to make it fail) or as a way to trigger a rollback of a running application. This second scenario will be described in the section 5.2.
Some tools like Sonarqube provide the possibility to create a quality gate through a graphic interface, while for some other tools it's necessary to explicitly analyse the content of the generated report.

One of the possible ways to implement a quality gate is based on the risk level associated to each vulnerability.
Usually an analysis tool generates a report where each vulnerability is identified by a specific name, a CVE or CWE, and it is associated to a risk level, frequently an integer or a string.

In order to check the exposure of the application, given a report we can count the number of vulnerabilities for each risk level.
Quality gate configurations can exploit the evaluation of different levels, each one with also a different threshold.

To apply this approach I identified three simple phases:

1. The first step is to choose, if possible, the format of the report. In my opinion this phase is also quite important and I suggest to choose an available report format which is quite minimal, without elements related to a graphic

environment.
Non-necessary elements will be just an additional noise factor that will make report analysis longer and more difficult. For this reason HTML and similar formats may be not the best choice.

2. The second step is to "study" the report structure in order to find a way to identify the vulnerability risk level.

3. The third step is to count the occurrences number.

This last phase depends on the complexity of the report and on the content itself. There are at least two possible approaches: the first one is based on the use of scripts that allows to perform complex parsing operations in a relatively simple way, while the second one is based on finding a specific string format. This second approach will be used in the following ZAP quality gate.
An important aspect related to the third step is that, if the count is based on strings, the format of the string used to identify the risk level has to be unique. This means that it must not be found in locations of the report that are different from those that we want to count, otherwise the occurrences number will be misrepresented.

These approaches are both enough efficient since the generated reports are usually quite small even if they contains thousands of items, allowing to perform also more scans in few seconds with a not so powerful hardware.

Below is reported a Python script that can be easily adapted to any kind of report by introducing just the counting of vulnerabilities, which is strictly related to the structure of the analysed report.
Moreover, is proposed also a Jenkins pipeline stage that uses the Python script, implementing a real quality gate. In this specific case if the quality gate is not passed, the pipeline will fail, but it is possible to configure it in order to perform other actions, for example the rollback of an application.

```python
import sys
#These values are compared with the number of vulnerabilities found, with the linked risk level.
#If the occurrences are more than the threshold, the script return a not-zero values.
lowThreshold=sys.argv[1]
mediumThreshold=sys.argv[2]
highThreshold=sys.argv[3]
criticalThreshold=sys.argv[4]
#
# Vulnerabilities counting, performed differently according to the report format
# In this phase the following variables are initialized: lowVuln, medVuln, highVuln, critVuln
#
#Quality gate
if (lowVuln>lowThreshold and lowThreshold!=-1): # -1 -> no threshold
        print("QG_FAILED")
        sys.exit(0)
else:
    if(medVuln>mediumThreshold and mediumThreshold!= -1):
            print("QG_FAILED")
            sys.exit(0)
    else:
        if(highVuln>highThreshold and highThreshold!= -1):
                print("QG_FAILED")
                sys.exit(0)
        else:
            if(critVuln>criticalThreshold and criticalThreshold!= -1):
                    print("QG_FAILED")
                    sys.exit(0)
            else:
                print("QG_PASSED")
                sys.exit(0)
```

Figure 5.1: Generic quality gate script in Python

```
pipeline {
    agent any
    stages {
        stage('QualityGate') {
            steps {
                sh "curl -LO https://example/qualityGate.py"
                script{
                   QG_RESULT= sh (script: """python3 qualityGate.py $lowVuln $medVuln $highVuln $critVuln"""
                   ,returnStdout:true).trim()
                   if(QG_RESULT=="QG_FAILED")
                       currentBuild.result = 'FAILURE'
                }
            }
        }
    }
}
```

Figure 5.2: Pipeline quality gate step with Python script

5.1.1 Quality gate in ZAP

In this section is proposed a simple way to implement a quality gate for a ZAP report, quality gate based on the number of vulnerabilities marked with a given risk level.

Following the three basic principles previously described, I started looking for available report formats in the official ZAP documentation and among them I decided to work and to apply this kind of quality gate on the .xml formatted file. I have taken this decision even if some kinds of report provide by default an overall view about the total number of found vulnerabilities, such as in the case of the html, because these reports are not easy to parse.
During the exploration of the xml report structure, I noticed that each vulnerability is characterized by two strings:

- A string of the type: `<riskcode>riskLevel</riskcode>`, where riskLevel is an integer number from 1 to 4, higher is the number more risky is the vulnerability: 1 = Low, 2=Medium, 3=High, 4=Critical.

- A string of the type: `<riskdesc>riskLevel(confidence)</riskdesc>`, where riskLevel and confidence are strings that can be "Low","Medium", "High" and "Critical". Moreover, the confidence value can be also "False Positive".

(The alert structure is reported in figure 4.11)

Once the string is identified, the last step is to count its occurrences. This last part can be performed in different ways according to the current scenario. For example it is possible to use a script or exploiting OS functionalities/commands such as grep for a Linux environment.

For instance, suppose that we decide that the quality gate has to fail if the number of critical vulnerabilities found is higher than 1.
Taking into account that "critical" is associated to the risk code "4", the resulting grep command applied to a ZAP .xml report would be:

```
grep -c "<riskcode>4</riskcode>" zapReport.xml
```

As extension of the previous described solution for the management of false positives in ZAP: if we want to take all the vulnerabilities that are not false positive, for a specific risk level, a command like the following one can be used:

```
grep -c -E "<riskdesc>Critical \([A-E]*[G-Z]*" zapReport.xml
```

These commands print on the standard output just the number of vulnerabilities with a Critical risk level.
Below is shown a tested pipeline step where this kind of quality gate is used.

The test is performed on Google Firing Range [20], a vulnerable web application maintained by Google. In this case the quality gate has to fail if there is at least a vulnerabilities marked with a medium risk level or a vulnerability marked with an high risk level.
Watching the results provided in the table 6.13, at the end of this step the variable QG_FAILED will assume the null value because at least one of the conditions is satisfied. Further steps will act according to the value of QG_FAILED.

```yaml
env:                                # The environment, mandatory
  contexts :                        # List of 1 or more contexts, mandatory
    - name: "test.context"          # Name to be used to refer to this context in other jobs, mandatory
      urls:  ["https://public-firing-range.appspot.com/"] # A mandatory list of top level urls,
                                    #everything under each url will be included
  vars:                             # List of 0 or more variables,
                                    #can be used in urls and selected other parameters
  parameters:
    failOnError: false              # If set exit on an error
    failOnWarning: false            # If set exit on a warning
    progressToStdout: true          # If set will write job progress to stdout
jobs:
  - type: spider                    # The traditional spider - fast but doesnt handle modern apps so well
    parameters:
      context: "test.context"       # String: Name of the context to spider, default: first context
      user:                         # String: An optional user to use for authentication, must be defined
                                    # in the env
      url: "https://public-firing-range.appspot.com/" # String: Url to start spidering from, default:
                                    # first context URL
      maxDuration:                  # Int: The max time in minutes the spider will be allowed to run for,
                                    # default: 0 unlimited
      maxDepth: 6                   # Int: The maximum tree depth to explore
      maxChildren:                  # Int: The maximum number of children to add to each node in the tree
  - type: activeScan                # The active scanner - this actively attacks the target so should only
                                    # be used with permission
    parameters:
      context: "test.context"       # String: Name of the context to attack, default: first context
      user:                         # String: An optional user to use for authentication, must be defined
                                    # in the env
      policy:                       # String: Name of the scan policy to be used, default: Default Policy
      maxRuleDurationInMins:        # Int: The max time in minutes any individual rule will be
                                    # allowed to run for, default: 0 unlimited
      maxScanDurationInMins:        # Int: The max time in minutes the active scanner will be
                                    # allowed to run for, default: 0 unlimited
  - type: report                    # Report generation
    parameters:
      template: "traditional-xml"   # String: The template id, default : modern
      theme:                        # String: The template theme, default: the first theme defined
                                    # for the template (if any)
      reportDir: "/zap/wrk"         # String: The directory into which the report will be written
      reportFile: "reportFiringRange"  # String: The report file name pattern,
                                    #default: {{yyyy-MM-dd}}-ZAP-Report-[[site]]
      reportTitle: "Report title"   # String: The report title
      reportDescription: "Description" # String: The report description
```

Figure 5.3: ZAP configuration file c.yaml

```
stage('ZAP analysis and Quality gate') {
    steps {
        dir('app') {
            //Download AutomationFramework file
            //It has to be download in the folder that will be mounted in Docker (e.g tmpDirectory)
            sh "curl -LO https://${DOWNLOAD_LINK}/c.yaml"
            //Execute ZAP analysis
            sh (script: "docker run -v ${tmpDirectory}:/zap/wrk/:rw -t owasp/zap2docker-stable:latest
                bash -c 'zap.sh -cmd -addonupdate; zap.sh -cmd -autorun /zap/wrk/c.yaml' -d -I", returnStdout:true)
            // Copy the generated report in the working Jenkins directory, in order to archieve it
            sh "cp ${tmpDirectory}/${ZAP_REPORT}  /var/jenkins_home/workspace/${folder}/app/${ZAP_REPORT}"
            archiveArtifacts artifacts: "**"
            //Quality Gate
            MEDIUM=sh (script: "grep -c '<riskcode>2</riskcode>' ${ZAP_REPORT} || true", returnStdout: true).trim()
            HIGH==sh (script: "grep -c '<riskcode>3</riskcode>' ${ZAP_REPORT} || true", returnStdout: true).trim()
            if(MEDIUM.toInteger()>0 || HIGH.toInteger()>0 ){QG_FAILED=true}
            else{QG_FAILED=null}
        }
    }
}
```

Figure 5.4: ZAP xml report - Pipeline ZAP test step

5.1.2 Quality gate with DefectDojo

A more interesting way to define a **generic** quality gate is to exploit the features provided by a vulnerability management tool, such as DefectDojo.
If we consider an `engagement` as one set of tests performed on the same code, then in an engagement we can have all the vulnerabilities found, by maybe different tools, on the actual version of the product.
It is possible to retrieve and filter these vulnerabilities, also looking for just those that are not false positives.
Thanks to this approach, just the correct/selected findings are considered in the quality gate evaluation.
Moreover, whenever vulnerabilities are imported into the vulnerability management tool they assume all the same format even if they are generated by different tools.
In a scenario where all the vulnerabilities have the same structure, quality gate analysis are easier.
Note that due to a different structure, to analyze a DefectDojo report a different command has to be used with respect to the previous one tested in figure 5.4.

To test this feature consider to upload the report from the pipeline as shown in the section 4.3.
Once DefectDojo contains all the vulnerabilities is possible to perform the quality gate stage as described in the image below, without requiring any manual actions.

```
stage('Quality gate'){
    steps{
        //DD_URL = URL to DefectDojo
        //AUTHORIZATION = Token to interact with DefectDojo
        //PRODUCT_ID = Integer ID assigned by DefectDojo when the
        //             product is inserted in the database
        script{
        //Get all the findings that are not duplicated or false positives
            sh """ curl -X GET "$DD_URL/api/v2/findings/?false_p=false&
            duplicate=false&test__engagement__product=$PRODUCT_ID&
            prefetch=&related_fields=true" -H  "accept: application/json"
            -H "Authorization: $AUTHORIZATION" > vulns.txt"""

            CRIT=sh(script: """grep -o '"severity":"Critical"' vulns.txt | wc -l """,
            returnStdout:true).trim()

            HIGH=sh(script: """grep -o '"severity":"High"' vulns.txt | wc -l """,
            returnStdout:true).trim()

            MED=sh(script: """grep -o '"severity":"Medium"' vulns.txt | wc -l """,
            returnStdout:true).trim()

            LOW=sh(script: """grep -o '"severity":"Low"' vulns.txt | wc -l """,
            returnStdout:true).trim()

            INFO=sh(script: """grep -o '"severity":"Info"' vulns.txt | wc -l """,
            returnStdout:true).trim()
            //QUALITY GATE - EXAMPLE
            //Suppose that the quality gate has to fail if there are at least
            //2 vulnerabilities with high risk or at least 1 marked as critical
            if(HIGH.toInteger()>1 || CRIT.toInteger()>0){
                currentBuild.result = 'FAILURE'
            }
        }
    }
}
```

Figure 5.5: Quality gate pipeline with DefectDojo

5.2 Rollback

In the real world, there are a near infinite number of possible deployment environments, each one with different features and components.

In this part of the thesis I have considered the environment where all the work was done: a simple Kubernetes cluster.
This topic can be strictly related to the previous one (quality gates), when the tested software is a running application.

Since this section is based on a particular environment, two assumptions has to be done:

1. The most important assumption is that we are working on a development environment. This means that if the application service faces some interruptions and become temporary not accessible, it will not be a problem; in a production scenario this could not be the same.

2. I suppose that a previous version of the application to test is already running. This previous version is considered safe, which means that it have passed the established quality gate. This second assumption is not essential, but I assume it to create a bit more characterized environment.

The approach used to implement the rollback functionality is composed by six steps. Steps 5 and 6 are executed only if the quality gate is not passed and so rollback has to be performed.

1. Stop of the previous version of the application: pod and service are deleted.
2. Start of the newer version of the application: new pod and service are created.
3. Dynamic analysis is performed over the new version of the application.
4. Quality gate.
5. Stop of the newer version of the application: pod and service are deleted.
6. Restart of the previous version of the application: new pod and service are created.

Some considerations are necessary: due the fact that through these commands I am not able to specify the port for the service, a bash script to retrieve this

information is necessary: without it I would not be able to perform the dynamic analysis since port would be unknown.
Moreover, we have to take into account that Jenkins pod has to interact with the Kubernetes cluster, so a proper configuration phase is needed otherwise kubectl commands would fail. I had to create two Kubernetes resources: `Role` and `RoleBinding`, necessary to allowing Jenkins pod to read and manipulate cluster pods and services. Below is shown a Jenkins pipeline with the stages previously described.

```
stages {
    stage('Build new image'){
        steps{
            script{
                sh "curl -LO ${DOCKERFILE_URL}"
                sh "docker build -t ${IMAGE_NAME}:${LAST_VERSION}"
            }
        }
    }
    stage('New version deployment'){
        steps {
            script{
                //HERE POD AND SERVICE ASSOCIATED TO THE PREVIOUS VERSION ARE DELETED
                sh (script: """kubectl delete pod '${POD_NAME}-v${PREVIOUS_VERSION}' """, returnStdout: true)
                sh (script: """kubectl delete service '${POD_NAME}-v${PREVIOUS_VERSION}' """, returnStdout: true)

                //NEW VERSION DEPLOYMENT

                //NEW POD CREATION
                sh (script: """kubectl run '${POD_NAME}-v${LAST_VERSION}' --image='${IMAGE_NAME}:${LAST_VERSION}'
                        --image-pull-policy=IfNotPresent""", returnStdout: true)
                //NEW SERVICE CREATION (ON A RANDOM PORT)
                sh (script: """kubectl expose pod '${POD_NAME}-v${LAST_VERSION}' --port=8080 --target-port=8080
                        --name='${POD_NAME}-v${LAST_VERSION}' --type=NodePort""", returnStdout:true)
                //RETRIEVE SERVICE PORT NUMBER. IT WILL BE USED DURING THE DYNAMIC ANALYSIS
                SERVICE_PORT=sh (script: """ kubectl describe service '${POD_NAME}-v${LAST_VERSION}' |
                        grep NodePort: | awk '{printf \$3}' | awk -F '/' '{printf \$1}' """, returnStdout:true)
            }
        }
    }
```

```
stage('ZAP analysis and Quality gate') {
    steps {
        dir('app') {
            //Download AutomationFramework file
            //It has to be download in the folder that will be mounted in Docker (e.g tmpDirectory)
            sh "curl -LO https://${DOWNLOAD_LINK}/c.yaml"
            //Execute ZAP analysis
            sh (script: "docker run -v ${tmpDirectory}:/zap/wrk/:rw -t owasp/zap2docker-stable:latest
                bash -c 'zap.sh -cmd -addonupdate; zap.sh -cmd -autorun /zap/wrk/c.yaml' -d -I", returnStdout:true)
            // Copy the generated report in the working Jenkins directory, in order to archieve it
            sh "cp ${tmpDirectory}/${ZAP_REPORT}  /var/jenkins_home/workspace/${folder}/app/${ZAP_REPORT}"
            archiveArtifacts artifacts: "**"
            //Quality Gate
            MEDIUM=sh (script: "grep -c '<riskcode>2</riskcode>' ${ZAP_REPORT} || true", returnStdout: true).trim()
            HIGH==sh (script: "grep -c '<riskcode>3</riskcode>' ${ZAP_REPORT} || true", returnStdout: true).trim()
            if(MEDIUM.toInteger()>0 || HIGH.toInteger()>0 ){QG_FAILED=true}
            else{QG_FAILED=null}
        }
    }
}
```

```
    stage('Rollback'){
        when{expression {return (QG_FAILED)} }
        steps{
            //QUALITY GATE IS NOT PASSED

            //DELETE POD AND SERVICE RELATED TO THE NEW VERSION
            sh (script: """kubectl delete pod '${POD_NAME}-v${LAST_VERSION}' """, returnStdout: true)
            sh (script: """kubectl delete service '${POD_NAME}-v${LAST_VERSION}' """, returnStdout: true)
            //VULNERABLE IMAGE IS DELETED
            sh ("docker image rm '${IMAGE_NAME}:${LAST_VERSION}'")
            //POD AND SERVICE ASSOCIATED TO THE PREVIOUS VERSION ARE CREATED
            sh (script: """kubectl run '${POD_NAME}-v${PREVIOUS_VERSION}' --image='${IMAGE_NAME}:${PREVIOUS_VERSION}'
                        --image-pull-policy=IfNotPresent""", returnStdout: true)
            sh (script: """kubectl expose pod '${POD_NAME}-v${PREVIOUS_VERSION}' --port=8080 --target-port=8080
                        --name='${POD_NAME}-v${PREVIOUS_VERSION}' --type=NodePort""", returnStdout:true)
        }
    }
}
```

Figure 5.6: Rollback pipeline test

Note that the c.yaml file used during the the ZAP analysis can be the same proposed in the figure 5.3

Another possible way to perform a rollback is to customize in an automated way a Kubernetes Deployment yaml file according to the new application specification.
This task may be not so easy when the Deployment file runs more containers each one with a different image and a different name.

Chapter 6

Tool testing, evaluation and limits

This chapter is split in five sections, each one related to a different type of tool: SAST, SCA, Container security, DAST and IAST.
Each section will contain analysis results, represented through tables. In this way it will be possible to have a more clear idea about how evaluated tools work and which are the differences.
Moreover, each section tries to identify some limitations associated to the testing phase and tries to give some conclusions about the results.

The initial idea to perform the evaluation was the analysis of benchmarks with known and well described vulnerabilities for each of the previously named tool types. In this way it would have been possible to compare true positives, false positives, true negatives and false negatives in a precise and objective way, giving a more honest result. However, just one suitable benchmark, and just for SAST tools, have been found. All the other benchmarks checked (e.g NIST SARD [21]) have not been used for this purpose since they had not a usable structure or were not written for a language supported by all the analysed tools.

It is important to take into account that different tools may evaluate vulnerabilities in a different way, which means in practice a vulnerability found by a first tool can have different values, such as the risk level, if compared with the result of a second tool; this is due the fact that tools may rely on different databases.

Note: In the evaluation sections, the names of the tools are substituted by placeholder. These placeholders are repeated over different sections, but the tools used may not be the same.

6.1 Test environment

The test environment is composed by a Minikube Kubernetes cluster (v1.25.2) which is executed on just a single node.
This cluster runs inside an Ubuntu:20.04 machine with 10GB RAM and 100GB of storage and Java 11.0.16 .
The Jenkins image, the core of the entire system, is provided by default with Java 11.0.15. In order to exploit Docker from the inside of the Jenkins container, I had to provide the proper permissions and I had to pass the Docker socket from the Minikube environment. To do this I mounted the socket through the deployment yaml file. Without it Jenkins would not have been able to start and manipulate Docker containers.

Standalone tools deployed in the cluster are: Sonarqube, DefectDojo and Arachni.
Standalone tools executed as Docker container from inside Jenkins are: ZAP and StackHawk.
Tools installed inside the Jenkins container are: CodeQL, Snyk (for SAST and for Container security), Grype, Trivy. DependencyCheck and Snyk (SCA) are executed through the Jenkins plugin.

Moreover, some tools require support software to work properly, software such as Java, Node, npm ... and for this reason having a custom Jenkins images containing these extra requirements could be a convenient solution.
The table 6.1 shows the version of the tools and the version of the required support utilities used inside the Jenkins container:

Tool	Version
Node	16.17.0
npm	8.15.0
Snyk	1.1006.0
Grype	0.48.0
Trivy	0.27.1
Sonarqube	8.9.1 Community Edition
DependencyCheck	7.1.1
CodeQL	2.10.1
DefectDojo	2.14.0
ZAP	2.11.1
Arachni	1.6.1.3-0.6.1.1
StackHawk	2.7.0

Table 6.1: Version of the tools

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jenkins
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jenkins
  template:
    metadata:
      labels:
        app: jenkins
    spec:
      containers:
      - name: jenkins
        image: jenkins/jenkins:lts-jdk11
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: jenkins-home
          mountPath: /var/jenkins_home
        - name: docker-socket
          mountPath: /var/run/docker.sock
      volumes:
      - name: jenkins-home
        emptyDir: {}
      - name: docker-socket
        hostPath:
          type: Socket
          path: /var/run/docker.sock
```

Figure 6.1: Jenkins deployment .yaml file

6.2 SAST tools analysis

In order to perform a comparison between the different SAST tools I used **OWASP Benchmark** [17].
The OWASP Benchmark Project is a test suite written in Java and created to perform the evaluation of vulnerability detection tools based on the accuracy, the speed and the coverage. These tools metrics allows to identify the weaknesses as well as than strengths of the analysed tools, allowing to perform more honest comparisons.
OWASP Benchmark is an open source web application that contains thousands of test cases, each one referring to a specific CWE. These test cases can be analysed by any type of Application Security Testing (AST) tool.
All the vulnerabilities deliberately included in this benchmark are exploitable, providing an additional way to evaluate the tools.
To be precise, the used version of this benchmark is composed by 2740 test cases, a quite small number if compared with the one of the previous version.
Here are reported the number of test cases for each possible vulnerability area.

Vulnerability Area	**# of tests**	**CWE Number**
Command Injection	251	78
Weak Cryptography	246	327
Weak Hashing	236	328
LDAP Injection	59	90
Path Traversal	268	22
Secure Cookie Flag	67	614
SQL Injection	504	89
Trust Boundary Violation	126	501
Weak Randomness	493	330
XPATH Injection	35	643
XSS (Cross-Site-Scripting)	455	79

Moreover, each Benchmark version comes with a spreadsheet that lists every test case, the vulnerability category, the CWE number and if the finding is a true positive or a false positive. This is very interesting because allows to evaluate tools in a more efficient way, allowing to compare four statistics strictly related to findings: True positives, False positives, True negatives and False negatives.
To perform the analysis on the OWASP benchmark, I used only the rules relatives to the CWE identified in the benchmark documentation. In this way I was able to

generate reports comparable with the expected benchmark results.

Here is reported the comparison between the three analyzed tools, enlightening the number of true positives and false positives.

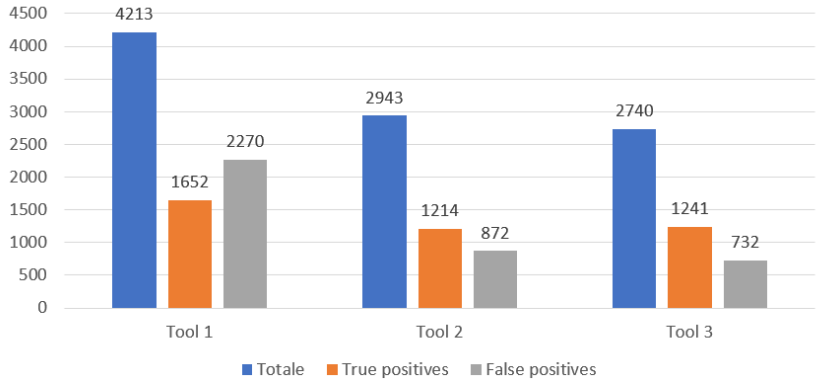

Figure 6.2: SAST tool results over OWASP Benchmark

More in details: here are reported the four parameters previously named, with the addition of the True Positive Rate [1] and of the False Positive Rate [2]:

Tool	Total	TP	FP	TN	FN	TPR	FPR
Tool 1	4213	1652	2270	230	61	0.964	0.908
Tool 2	2943	1214	872	656	201	0.857	0.570
Tool 3	2740	1241	732	593	174	0.877	0.552

Table 6.2: OWASP Benchmark SAST findings type

[1]TPR=TP/(TP+FN) . Is the probability that an actual positive will test positive

[2]FPR=FP/(FP+TN) . Is the ability to avoid reporting false errors

In order to have a graphical representation of the tools "quality", a possible approach is to use the following graph.
The tool quality will depend on the position inside this graph, based on the TPR and FPR values.

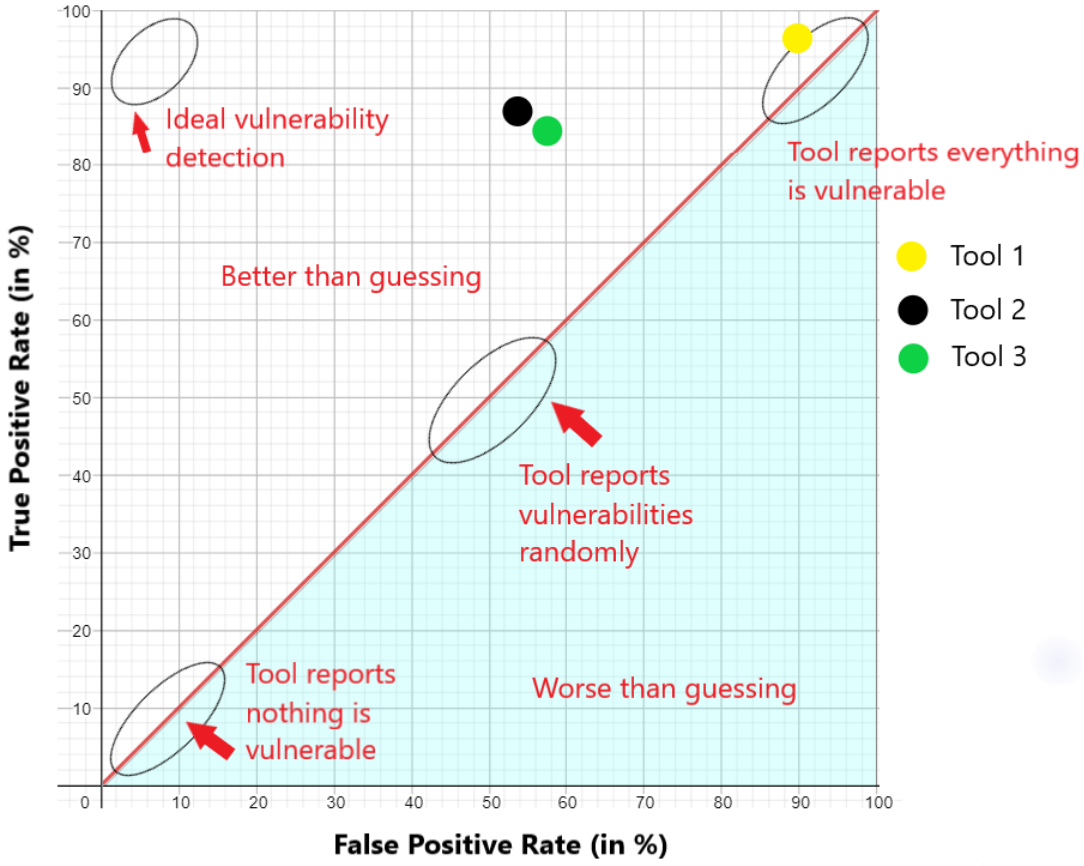

Figure 6.3: SAST Benchmark graphic result

In order to verify the results obtained in the previous benchmark analysis, following are reported some other analysis results, useful to better understand the way this three tools work and in particular to compare the total amount of findings, verifying if the trend related to the number of vulnerabilities shown in the first case is repeated also in these additional ones.

In particular, the analysis are performed over two projects: BioJava [3] and Tweety [4]

BioJava

Tool	Critical	High	Medium	Low	Info	Time
Tool 1	1	13	7	8	0	12min
Tool 2	0	12	0	1	0	5min
Tool 3	1	8	0	0	0	14min

Table 6.3: SAST analysis: BioJava

Tweety

Tool	Critical	High	Medium	Low	Info	Time
Tool 1	131	1118	2898	0	360	3min
Tool 2	0	11	1	2	0	6min
Tool 3	0	12	0	1	0	5min

Table 6.4: SAST analysis: Tweety 1

Tweety

Tool	Critical	High	Medium	Low	Info	Time
Tool 1	131	1118	2898	0	2147	3min
Tool 2	0	11	1	2	0	6min
Tool 3	0	12	0	1	0	5min

Table 6.5: SAST analysis: Tweety 2

Test Limitations

The usage of just one benchmark may provide a false idea about the quality of the analysed SAST tools. In order to have a more precise evaluation, the usage of more benchmarks that allow to compute the number of FP,TP, FN and TN would be a possible solution or at least a mitigation.

[3]Biojava is an open-source project dedicated to providing a Java framework for processing biological data [22]

[4]Tweety is a collection of various Java libraries that implement approaches to different areas of artificial intelligence [23]

As already specified in the introduction of this section, I was not able to test other benchmarks to evaluate in a deeper way the results: all the found benchmarks were not suitable to the current scenario because they were not written for supported languages or had not a supported structure.

Conclusions about SAST tools evaluation

In general, using Tool 1 I experimented that it's not useful to keep applied all the default rules for a specific language, because usually this leads to have a very high number of findings, as reported in the tables 6.4 and 6.5 where the difference is more or less 2000 findings: a very high number of vulnerabilities that a security expert should verify or at least should filter, processes that need time.
In the table 6.4 I performed a Tool 1 analysis applying only the rules relative to the OWASP TOP 10 classification (55 rules) while in the table 6.5 the analysis was performed by applying all the rules related to vulnerability and security hotspot classes (222 rules).

A similar test for Tool 2 and Tool 3 was not possible because one of them does not allow to change the applied rules, while for the other the only way to change this set is to create a new one with less rules, which is not a quite simple task and is not useful in this case since the result is already affordable.

Comparing SAST tools over OWASP Benchmark, the results evidence that Tool 1 reports an higher number of false positives if compared with Tool 2 or with Tool 3. It is important to note that this happens even if have been activated only the rules useful to find the well-known vulnerabilities inside the benchmark.
Moreover, using TPR and FPR values, and referring to the graph 6.3 where the analysed tools were positioned, it is possible to notice that Tool 1 take place more or less in the area where contained tools are categorized as reporting everything is vulnerable.
Differently, Tool 2 and Tool 3 take place in a different area, associated to an overall higher quality.
Since Tool 1 seems to be a tool that even when using a restricted set of rules finds many false positives, is possible that if a more generic set of rules is applied (e.g. to analyse projects with unknown vulnerability types) there will be more false positives, and this leads to the security experts to lose a lot of time to verify a lot of fake alerts.
This idea is corroborated also by the analysis results reported in the tables 6.3, 6.4 and 6.5, where it is possible to observe the difference between the results generated by Tool 2 and Tool 3 from those generated by Tool 1.

With regard to Tweety analysis, which is a bigger and more complex application than the BioJava project, we can see that the Tool 1 number of findings is more than 1000 time higher than the number of findings generated by the other two tools.

Anyway, in addition to numerical statistics, we have to take into account also that one of this tools needs to interact with its servers during the whole analysis, so it has to be connected to Internet. Moreover, the free version allows just a limited number of custom code scans; in an enterprise scenario an upgrade could be necessary.

6.3 SCA tools analysis

Differently from the previous case, I did not find a benchmark to test SCA tools and in order to obtain some result about how this kind of tools work, I analyzed some different open source project trying to compare the results.
The analysed projects are:

- OWASP Open Juice [24]
- BioJava [22]
- Tweety [23]

OWASP Open Juice

Tool	Critical	High	Medium	Low	Info	Time
Tool 1	4	16	15	3	0	1min
Tool 2	2	1	26	0	0	20min

Table 6.6: Software Composition Analysis: OWASP Open Juice

BioJava

Tool	Critical	High	Medium	Low	Info	Time
Tool 1	0	2	1	1	0	4min
Tool 2	0	0	2	0	0	7min

Table 6.7: Software Composition Analysis: BioJava

Tweety

Tool	Critical	High	Medium	Low	Info	Time
Tool 1	0	0	42	2	0	6min
Tool 2	1	11	4	0	0	4min

Table 6.8: Software Composition Analysis: Tweety

Test Limitations

Due to the impossibility to find a suitable benchmark, I was not able to make a reasoning based on false positives as done for SAST tools; this limitation does not allow to really understand the quality of the reported vulnerabilities.

The only way that we have to compare these results is the number of findings and the time spent to perform the analysis.

Conclusions about SCA tools evaluation

Both tools do not check if the libraries are really used, they just check if these elements are imported. In order to verify this behaviour I have created a project composed by just manifest files, without source code. In both the cases, tools have been reported vulnerabilities.
A second test to check how the analysis work, was performed by introducing the source code next to the manifest files: for both the tools the results do not change.
Analysing the number of findings provided in the tables 6.6, 6.7 and 6.8, seems that Tool 1 is able to find an higher number of vulnerabilities compared to Tool 2. This happens also because one of the tools stops the analysis across a branch when a vulnerable dependency is found (described in the section 6.3), while the other tool does not work in this way.

Conversely, the time spent to analyse the same codebase seems to be higher for Tool 2 than for Tool 1.

Furthermore, it is important to take in mind that, also in this case, DependencyCheck runs in local and needs a internet connection just for the updates while Snyk needs to be connected to its servers to perform analysis.

6.4 Container Security tools analysis

This kind of analysis was performed analysing common and public Docker images. It's important to remind that an image downloaded from a not official or not trusted repository may contains even more vulnerabilities than the official one, since it may have been manipulated.
Analysed images are:

- ubuntu:22.04 : Ubuntu is an open source operating system on Linux for the enterprise server, desktop, cloud, and IoT. [25]

- openjdk:8-jre-alpine: OpenJDK (Open Java Development Kit) is a free and open-source implementation of the Java Platform, Standard Edition (Java SE). [26]

- wordpress:latest: Wordpress is a free and open-source content management system (CMS) written in PHP and paired with a MySQL or MariaDB database. [27]

- centos:centos8 : CentoOS is a Linux distribution that provides a free and open-source community-supported computing platform, functionally compatible with its upstream source, Red Hat Enterprise Linux (RHEL). [28]

These analysis have been performed with the default rules for each tool.
Below are reported the results for each analysed image.

Ubuntu:22.04

Tool	Critical	High	Medium	Low	Info	Total #	Time
Tool 1	0	0	3	15	0	18	12s
Tool 2	0	0	2	7	8	17	7s
Tool 3	0	0	2	19	0	21	13s

Table 6.9: Image analysis: Ubuntu

openjdk:8-jre-alpine

Tool	Critical	High	Medium	Low	Info	Total #	Time
Tool 1	4	27	79	106	0	216	11s
Tool 2	15	55	93	108	0	271	8s
Tool 3	4	27	79	106	0	216	11s

Table 6.10: Image analysis: openjdk

wordpress:6.0.2

Tool	Critical	High	Medium	Low	Info	Total #	Time
Tool 1	6	69	92	364	0	531	11s
Tool 2	6	69	92	13	347	527	7s
Tool 3	4	1	0	438	0	441	12s

Table 6.11: Image analysis: Wordpress

centos:centos8

Tool	Critical	High	Medium	Low	Info	Total #	Time
Tool 1	0	22	208	132	0	362	14s
Tool 2	0	24	208	116	4	352	9s
Tool 3	0	31	213	129	0	373	15s

Table 6.12: Image analysis: CentOS

Test Limitations

The most serious limitation with these analysis is the lack of distinction between true positive and false positives, which does not allow to really understand the quality of the reported vulnerabilities.
The only way that we have to compare these results is the number of findings and the time needed to perform the scans.

Conclusions about Container Security tools evaluation
Looking the results obtained by the analysis performed over these four images, they show that the three compared tools are able to find, more or less, the same quantity of vulnerabilities.
Going deeper, by comparing the CVEs identified in each report, the results can be almost overlapped. This comparison was performed by taking into account the CVEs and the paths where the vulnerabilities have been found, in order to avoid the possible scenario where the same CVE is identified in different position.
The time needed to perform these analysis, without including the time to download the images, is more or less the same between the different tools, just few seconds of difference.

So, given the shown results, it seems that these tools are able to find similar or even the same vulnerabilities in the same time; the only difference is that Snyk is a commercial tool that offers unlimited scan for open source vulnerabilities (so for SCA and for container security) but needs to be constantly connected to Internet, while Trivy and Grype are able to work offline.

6.5 DAST tools analysis

In order to test DAST tools I used local deployed applications with known vulnerabilities and vulnerable web application, publicly exposed and created to be attacked.
The analysed applications are:

- Google Firing Range [20]: is a test bed for automated web application security scanners.
  The public instance is reachable at: https://public-firing-range.appspot.com/
- SecurityTweets [29]: a vulnerable by design web application publicly reachable at: http://testhtml5.vulnweb.com/
- OWASP Open Juice [24]: a vulnerable web application developed and mainteined by the OWASP team.
- Gruyere [30]: a Google vulnerable web application used to training purposes.

Here are reported the results divided for each tested application

Google Firing Range

Tool	Critical	High	Medium	Low	Info	Avg.Time
Tool 1	0	3	10	6	2	60min
Tool 2	0	3	7	7	0	33min
Tool 3	0	0	1	0	0	3min
Tool 4	0	0	2	2	2	75min

Table 6.13: DAST analysis: Google Firing Range

Gruyere

Tool	Critical	High	Medium	Low	Info	Avg.Time
Tool 1	0	2	3	5	2	7min
Tool 2	0	1	4	5	0	5min
Tool 3	0	0	2	4	0	20min
Tool 4	0	1	1	1	2	16min

Table 6.14: DAST analysis: Gruyere

OWASP Juice Shop

Tool	Critical	High	Medium	Low	Info	Avg.Time
Tool 1	0	0	2	2	1	10min
Tool 2	0	1	4	6	0	7min
Tool 3	0	0	0	2	0	2min
Tool 4	0	0	2	2	0	2min

Table 6.15: DAST analysis: OWASP Juice Shop

SecurityTweets

Tool	Critical	High	Medium	Low	Info	Avg.Time
Tool 1	0	0	5	4	1	7m
Tool 2	0	0	4	5	0	5m
Tool 3	0	0	0	6	0	4m
Tool 4	0	0	1	2	2	9m

Table 6.16: DAST analysis: SecurityTweets

Test Limitations

The initial idea was to dynamically test OWASP Benchmark since it is a runnable application.
This would be a quite good way to compare the results generated by DAST tools, but unfortunately this was not possible. As reported on the official guide of OWASP, the benchmark requires an high amount of resources to run, to which I had to add resources required by the DAST tools and by the Minikube cluster.
So, due to hardware limitations I had not the possibility to complete this analysis and comparison by using this benchmark.

Conclusions about DAST tools evaluation

As already said, we have to take into account that different tools may use different

vulnerability databases where to each vulnerability is assigned a different risk level. Analyzing these results and making a comparison, it is possible to say that not all the tools find the same vulnerabilities in the same time.
Tool 4 and Tool 3 seems to provide more approximately results, even requiring in some cases a longer analysis time.
On the other side, Tool 1 and Tool 2 seem to provide more accurately and better described reports, which are two essential requirements that allow security experts to check the findings in a easier and more efficient way.
Talking about the functionalities provided by the different tools, a positive aspect of StackHawk is that provides a quite useful and intuitive dashboard that allows also to repeat and validate through curl requests [31] all the found vulnerabilities, while the others just provide a non-interactive report.
However, StackHawk is a commercial tool and in its free version provides just a limited set of functionalities. For instance, it does not give the possibility to interact directly with a Vulnerability Management Tool and just one application at time can be monitored.

6.6 IAST evaluation - Contrast C.E.

Performing an evaluation on a single tool, without having parameters to refer to, it's a quite hard task. For this reason I will try to highlight which are its characteristics and the possibility of including this IAST tool inside a DevSecOps pipeline.

In order to test and to understand how this tool works, I used again OWASP Benchmark:

1. The first step was starting the Java benchmark application with the Contrast agent onboard, providing also a very simple configuration file filled with Contrast credentials.

2. After some seconds a first and incomplete result appeared in the dashboard (Contrast website), showing few results.

3. At this point the interactive phase started, which was performed through the launch of a script provided inside the benchmark itself: this script interacted with the running application triggering new code lines and providing a more complete scan result if compared with the previous one.
   This phase can be also performed with a manual approach, but in this way the automation is lost.

One of the most heavy limitation is related to the fact that this tool supports only few technologies and programming languages.
Moreover, Contrast C.E. analyses a running application and has to access at least to the part of code that will be executed; this means that before finding vulnerabilities developers has to create an executable application, retarding the analysis in a relatively advanced phase of the development.
An `agent` is bound to the execution environment in a way that depends on the used technology and has to interact with the Contrast servers during the whole analysis.

On the other side, Contrast provide at least two interesting features which are related to the dependencies and to the implementation of a quality gate.
Through the dashboard is possible to verify which dependencies are really used, potentially providing a more precise result than the previously showed SCA tools that check only if a dependency is imported.
A plugin for Jenkins is available to implement a quality gate: it retrieves the information about the application exposure to vulnerabilities from the Contrast server and then the quality gate is performed according to the plugin configuration. It can be be configured through the plugin or through the website, but it seemes to be not available in the free version.

The phases initially described to test OWASP benchmark can be summarized and adapted to the usage of Contrast C.E. inside a CI/CD Pipeline:

1. Application execution with Contrast agent.
2. Interactive phase: this can be done manually, but in an automated environment like a CI/CD pipeline this should not happen: we can use scripts/acceptance tests that trigger the code functionalities.
3. Quality gate: once the interactive phase is completed, quality gate can be performed. The current security exposure of the application is compared with the configured quality gate.
4. Pipeline end: the result of the quality gate is used to set the pipeline status as failed, completed successfully or completed but unstable.
   As alternative, rollback operations can be performed, also following the principles describe in the chapter 5.

As previously said, the Jenkins plugin seems to be not available in the free version, so I wasn't able to test this last feature.

Chapter 7

Conclusions and future works

The objectives of this thesis were: the management of false positives, in particular for DependencyCheck, ZAP and with DefectDojo; the introduction of quality gate for ZAP and with DefectDojo; the introduction of the rollback functionality and the research and evaluation of new tools, everything inside a DevSecOps scenario characterized by the usage of CI/CD pipelines.

The environment where this work was done is a quite simple Kubernetes cluster created through Minikube, which allowed to deploy tools and tested applications.

With respect to the management of false positives, in the company initially solutions were not yet applied, but both for DependencyCheck and ZAP a way to introduce this feature was found. In particular for ZAP more possible paths have been discussed but just one of them, in the actual state, works as expected.

As in the previous case, even in the introduction of quality gate for ZAP no solutions were initially available.
The described implementation of the quality gate was based on the counting of vulnerabilities for each risk level, counting performed over a .xml formatted report file.
For both these two objectives, an additional solution for each task was implemented with DefectDojo, a vulnerability management tool, in order to provide a more generic approach that could be applied also to other tools.

Considering the current Kubernetes development scenario, a solution for the rollback of an application based on the management of Pods and Services was

proposed: given a running applications, if the quality gate fed with the results produced by a DAST tool (e.g. ZAP) over this application fails, then newly created Pods and Services are stopped while old ones related to the previous and considered safe version are re-created.

The searching of the new tools was performed under some particular guide lines identified in the section 3.3 while the evaluation was performed in different way according to the different kind of tool. For SAST tools a benchmark was used while for SCA, Container security and DAST tools were used respectively open source projects, images and open source applications.
Talking about the results, it seems that for SAST, SCA and DAST analysis, some of the evaluated tools performs better than the others, while seems that analysed Container security tools are able to find more or less the same vulnerabilities in the same time.

FUTURE WORKS

Quality gate chapter could be completed with the creation of quality gates also for other tools; this means that their report has to be analysed and a way to count vulnerabilities has to be found. Moreover, new approaches to create quality gates can be found.

As already proposed at the end of the section 5.2, it would be possible to explore the rollback functionality when Deployment yaml files are used, so changing the way Pods and Services are handled. This means that a new Kubernetes resource has to be used with all its particular features and characteristics.

In my opinion, one of the most important possible improvements is related to the evaluation of the selected tools and in particular is the analysis of more benchmarks. More benchmark would be useful to better describe the quality of the tools and so to give a more honest view of their characteristics.
As already said, using just one benchmark may leads to have a limited overview of the different tools behaviour, producing not reliable conclusions.

Another interesting aspect would be the discovering of others new tools, especially those which regards to the IAST approach.

Bibliography

[1] *Security perimeter definitions*. URL: https://csrc.nist.gov/glossary/term/security_perimeter (cit. on p. 6).

[2] *6 BENEFITS OF THE DEVSECOPS MODEL*. URL: https://snyk.io/series/devsecops/ (cit. on p. 6).

[3] *CI/CD Pipeline: A Gentle Introduction*. URL: https://semaphoreci.com/blog/cicd-pipeline (cit. on p. 9).

[4] *DevSecOps Pipeline - A Complete Overview | 2022*. URL: https://www.xenonstack.com/insights/guide-devsecops-pipeline (cit. on p. 9).

[5] *Application security testing: come orientarsi fra SAST, DAST o IAST?* URL: https://www.zerounoweb.it/software/application-security-testing-come-orientarsi-fra-sast-dast-o-iast/ (cit. on pp. 11, 17).

[6] *SCA benefits*. URL: https://www.geeksforgeeks.org/overview-of-software-composition-analysis/ (cit. on p. 12).

[7] *What is Container Security?* URL: https://www.vmware.com/topics/glossary/content/container-security.html (cit. on p. 13).

[8] *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/ (cit. on p. 14).

[9] *Interactive Application Security Testing (IAST)*. URL: https://snyk.io/learn/application-security/iast-interactive-application-security-testing/ (cit. on p. 17).

[10] *All About IAST – Interactive Application Security Testing*. URL: https://www.mend.io/resources/blog/iast-interactive-application-security-testing/ (cit. on p. 17).

[11] *What is a Kubernetes*. URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/ (cit. on p. 23).

[12] *SonarQube docs*. URL: https://docs.sonarqube.org/ (cit. on p. 26).

[13] *CPE*. URL: https://nvd.nist.gov/products/cpe (cit. on p. 27).

[14] *What is a CVE?* URL: https://www.redhat.com/en/topics/security/what-is-cve (cit. on pp. 27, 39).

[15] *ZAP Automation Framework.* URL: https://www.zaproxy.org/docs/automate/automation-framework/ (cit. on p. 28).

[16] *NVD website.* URL: https://nvd.nist.gov// (cit. on p. 31).

[17] *OWASP Benchmark Project.* URL: https://owasp.org/www-project-benchmark/ (cit. on pp. 38, 68).

[18] *Dependency Check - Suppressing False Positives.* URL: https://jeremylong.github.io/DependencyCheck/general/suppression.html (cit. on p. 41).

[19] *Managing False Positives in OWASP Zed Attack Proxy (ZAP).* URL: https://jiarongchew.medium.com/managing-false-positives-in-owasp-zed-attack-proxy-zap-a2581e64c249 (cit. on p. 48).

[20] *GoogleFiringRange GitHub.* URL: https://github.com/google/firing-range (cit. on pp. 59, 78).

[21] *NIST SARD - Test suites.* URL: https://samate.nist.gov/SARD/test-suites (cit. on p. 65).

[22] *BioJava Project.* URL: https://github.com/biojava/biojava (cit. on pp. 71, 73).

[23] *TweetyProject.* URL: https://tweetyproject.org/ (cit. on pp. 71, 73).

[24] *OWASP Open Juice.* URL: https://owasp.org/www-project-juice-shop/ (cit. on pp. 73, 78).

[25] *Ubuntu.* URL: https://docs.ubuntu.com/ (cit. on p. 75).

[26] *OpenJDK.* URL: https://openjdk.org/projects/jdk8/ (cit. on p. 75).

[27] *Wordpress.* URL: https://wordpress.org/ (cit. on p. 75).

[28] *CentOS Project.* URL: https://wiki.centos.org/Documentation (cit. on p. 75).

[29] *Security Tweets.* URL: http://testhtml5.vulnweb.com/#/popular (cit. on p. 78).

[30] *Gruyere.* URL: https://google-gruyere.appspot.com/ (cit. on p. 78).

[31] *Linux Curl Command.* URL: https://linuxhint.com/linux-curl-command/ (cit. on p. 80).