FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dynamic Security Testing

João Pedro Marques Rôla

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Ricardo Morla

August 21, 2022

Resumo

A metodologia de desenvolvimento de *software DevOps* e a utilização de *pipelines* CI/CD aceleraram significativamente o tempo de entrega dos seus adoptantes, mas esta abordagem ao desenvolvimento de *software* coloca desafios à forma como a segurança é tradicionalmente implementada, por esta não ser capaz de acompanhar o ritmo do Ciclo de Vida do Desenvolvimento de *Software DevOps*. Torna-se assim, necessária uma nova abordagem sobre como integrar a segurança neste paradigma de desenvolvimento.

DevSecOps visa responder este problema incorporando técnicas de segurança nas práticas de *DevOps*, deslocando assim a segurança para as fases iniciais de desenvolvimento. Este processo promove a automação de testes para a simplificação de métodos que por outros meios seriam mais complexos e demorados. Actualmente, *DevSecOps* é um tópico popular entre profissionais do setor de desenvolvimento de *software*, como se pode ver pela produção de literatura cinzenta nesta área, mas não lhe é dada tanta atenção por parte dos meios académicos.

Esta dissertação apresenta um estudo sobre a forma de integrar funcionalidades de segurança, tais como testes de segurança dinâmicos a aplicações (DAST) e testes de segurança estáticos a aplicações (SAST), em *pipelines* CI/CD, e qual o impacto destas adições na segurança do código utilizado para construir artefactos de *containers*, no tempo para lançamento e na segurança da *pipeline* em si.

Para o conseguir, primeiro, foi criada uma arquitectura *pipeline* CI/CD de referência tendo em conta o caso particular de utilização. Em seguida, foi realizada uma análise dos relatórios de vulnerabilidade produzidos pelas ferramentas DevSecOps. Nesta análise, a eficácia e a eficiência dos testes, considerando as métricas de desempenho do sistema, foram avaliadas. Finalmente, fazendo uso da análise anteriormente mencionada e da documentação da arquitectura da *pipeline*, foram tiradas conclusões sobre os efeitos das fases de testes de segurança no processo de CI/CD.

Abstract

The DevOps software development methodology and the use of CI/CD pipelines have sped up the delivery time of its adopters significantly, yet this approach to software development poses challenges to the way security is traditionally implemented, by it not being able to keep up with the pace of the DevOps Software Development Life Cycle. Thus, a new approach on how to integrate security into this development paradigm is required.

DevSecOps aims to answer this problem by merging security techniques into DevOps practices, shifting security to the earlier stages of development. This process leverages testing automation for the simplification of otherwise more complex and time-consuming methods. Currently, DevSecOps is a popular topic with industry practitioners, as can be seen by the grey literature output of this field, but not as much attention has been given to it by academia.

This dissertation presents a study on how to integrate security features, such as dynamic application security testing (DAST) and static application security testing (SAST), into CI/CD pipelines, and what impact these new stages will have in the safety of the code used to build container artifacts, time to deploy and security of the pipeline itself.

In order to achieve this, first, a reference CI/CD pipeline architecture was created taking into account the particular use-case. Then an analysis of the vulnerability reports produced by the DevSecOps tooling, was performed. In this analysis, the effectiveness and efficiency of the tests, considering the system's performance metrics, was evaluated. Finally, making use of the previously mentioned analysis and pipeline architecture documentation, conclusions on the effects of the security testing stages on the CI/CD process were drawn.

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus orientadores, Prof. Ricardo Morla e André Duarte, pelo apoio prestado ao longo do desenvolvimento do trabalho. Sempre com disposição amigável, contribuíram com sugestões e comentários que me ajudaram a "ver a luz ao fundo do túnel" nestes últimos meses. Estou também grato à Ubiwhere por me ter dado a oportunidade de trabalhar nesta proposta que me cativou o interesse e me motivou a aprender com os desafios que me foram postos.

Queria também agradecer aos meus amigos que ao longo deste percurso me proporcionaram momentos de alegria que me lembrarei para a vida. À Carolina, que nesta nossa cumplicidade esteve sempre disposta a apoiar-me, mesmo quando a chateava com assuntos aborrecidos.

Ao meu irmão, pela melhor amizade que alguém pode ter. Aos meus pais e avós, por me terem providenciado tudo o que precisei para chegar aqui e por todo o encorajamento que me deram. E por fim, a toda a minha restante família, pelo afeto que me deram.

João Rôla

"We become what we behold. We shape our tools, and thereafter our tools shape us."

John Culkin

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

API	Application Programming Interface
CD	Continuous Delivery / Continous Deployment
CI	Continous Integration
CSRF	Cross-Site Request Forgery
DAST	Dynamic Application Security Testing
DevOps	Development and Operations
DevSecOps	Development, Security and Operations
DoS	Denial of Service
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IAST	Interactive Application Security Testing
IT	Information Technology
k8s	Kubernetes
OSS	Open Source Software
PaaS	Platform as a Service
SaaS	Software as a Service
SAST	Static Application Security Testing
SDLC	Software Development Life Cycle
SQLi	SQL Injection
XSS	Cross-Site Scripting

Chapter 1

Introduction

1.1 Context

Nowadays, we are seeing the widespread adoption of DevOps practices, this software development model although lacking a universal definition, can be identified by some core traits and values such as, a culture of increased cooperation between the IT operations and development teams, workflow automation, monitoring and observability and the concepts of Continuous Integration / Continuous Delivery.

The main driving force behind the shift to the DevOps model is the reduced time it takes for software changes to enter production, this gives its practitioners the ability to deploy new iterations of their products much faster than traditional models would allow. This is facilitated by Continuous Integration and Continuous Delivery practices. In addition to that, a substantial number of software development companies nowadays utilise Cloud Computing service models for their offerings, such as Software as a Service (SaaS), where software products are hosted in cloud infrastructure with no need for user installation of updates, giving developers the ability to update frequently without disrupting the end user experience.

This methodology, however presents new challenges to the implementation of security practices in the software development life-cycle as these are commonly implemented at the end of it and tend to be lengthy endeavours which slow down deployments. With this problem in mind, DevSecOps tries to integrate security practices into DevOps, by shifting security “to the left”, as in earlier in the development stage.

1.2 Motivation

With the aim of reconciliation of security testing and CI/CD in mind, new pipeline architectures and DevOps tools have been developed to make these two approaches compatible. To better understand how to incorporate these security features into DevOps, the design of CI/CD pipelines needs to be studied in order to assess how DevSecOps mechanisms, methods and tools, integrate into existing DevOps pipelines, this study of multiple possible configurations will be beneficial for the understanding of the effectiveness, scalability, and code security at the output of DevSecOps pipeline architectures. As this is not a popular topic in academia, at the moment, this effort will attempt to clarify and quantify the benefits and drawbacks of incorporating DevSecOps into a new project.

1.3 Objectives

The expected outcomes of this dissertation are the creation of a DevSecOps pipeline, and a study that documents the performance results, modularity, scalability and compatibility of each solution. The pipeline that will be discussed, is responsible for the generation of containerized artifacts, the security of each artifact will be tested through code analysis and application testing. The pipeline itself will be designed with its own security in mind, as well as time constraints so that multiple deployments per day are achievable.

1.4 Document Structure

The document is divided into six chapters and an appendix. Apart from this first chapter, the second chapter 2 provides a literature review on topics related to or that are going to be discussed in further chapters. The third chapter 3 details the specifications and considerations in the design of the CI/CD pipeline, from which time and vulnerability metrics were extracted. The fourth chapter 4 explains how these metrics were collected, offers an analysis of these and discusses the implications they have in a DevSecOps strategy. The sixth and last chapter 5, highlights the major conclusions of the work described in the previous chapters, and references possibilities for future work.

Chapter 2

Literature Review

2.1 Cloud Computing

Nowadays, the Cloud Computing model has become increasingly more accessible, public cloud providers can provide projects the means to implement their own cloud infrastructure, without having to resort to building their own, while also being feasible from a profitability perspective, when compared to an on-premises solution [3]. It also has created new service models for businesses and institutions, these have been made possible due to capabilities specific to cloud computing. Naturally, this has lead to an increase in interest in this approach, which has had an impact in the application development process.

Cloud Computing according to [4], is defined as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction.

This model has five main characteristics:

1. On-demand self-service: A consumer can provision computing resources automatically without requiring human interaction with a service provider.

2. Broad network access: Capabilities can be accessed over the network through standard mechanisms.

3. Resource pooling: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

4. Rapid elasticity: Capabilities can be provisioned and released elastically to provide scalability according to demand.

5. Measured service: The cloud system controls and monitors resource usage for optimization, transparency purposes.

The service models can be characterized as:

1. Software as a Service (SaaS): The consumer uses the provider's applications running on a cloud infrastructure. The consumer does not control the application's cloud infrastructure.

2. Platform as a Service (PaaS): The consumer deploys to the cloud provider's infrastructure, applications using the provider's development environment (i.e. the programming languages, libraries, services, and tools supported by the provider) The consumer does not have control over the underlying infrastructure.

3. Infrastructure as a Service (IaaS): The consumer can provision computing resources in which it can run software chosen by himself. The consumer does not have control over the cloud's infrastructure, yet it can choose operating systems, storage, deployed applications and networking options.

2.1.1 Cloud Native

Arisen out of the Cloud Computing paradigm, Cloud Native is an approach to development that tries to take advantage of the capabilities of cloud computing, such as those listed in section 2.1, typically through a Microservices architecture [5]. In a Microservices architecture, an application is broken down to several smaller independent services that run on containers, this approach facilitates Continuous Software Development by enabling parallel development of features. It provides better modularity, scalability at the cost of higher network costs and latency. These applications are commonly deployed in clusters, managed by container orchestration software, such as Kubernetes.

2.1.2 Microservices

Microservices are small applications with a single responsibility that can be deployed, scaled, and tested independently. It also enables agile teams to structure their work around these services, given that microservices are, by definition, autonomously developed [6]. Typically used in Cloud Native / DevOps applications, these provide benefits in availability (if well integrated), maintainability, testability at the cost of reliability, performance and complexity in implementing security, all of these properties have to do with the distributed nature of this type of architecture [6].

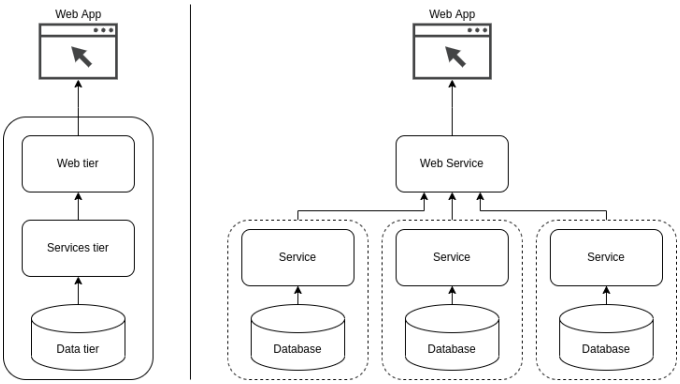

Figure 2.1: Monolithic and Microservices architecture

2.1.3 Containerization

Containers are a form of OS-level virtualization, they are primarily used for delivering software, that is, they have a platform-as-a-service (PaaS) focus [1], they also make it easier to move applications between development, testing, and production environments. By providing a lightweight portable runtime environment, containers let developers package and isolate their apps with everything they need to run, including application files, dependent libraries and configurations.

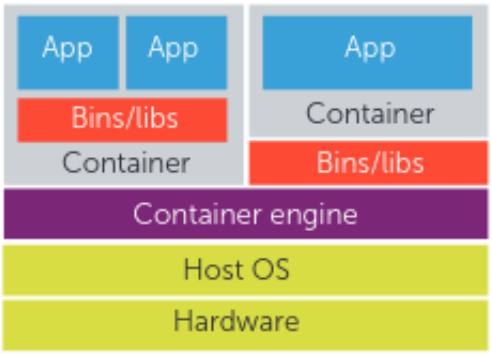

Figure 2.2: Containerization illustration from [1]

2.1.4 Infrastructure as Code

Infrastructure as Code (IaC) permits the use of "source code", for infrastructure designs, such that the entire set of scripts, automation and configuration code, models, required dependencies and operational configuration parameters can be expressed using the same standard language, version control could also be applied to these scripts, all these properties enable the merging of the concepts of operations and development. [7] Thus, by being less error-prone, and easier to test, it provides a faster and safer way to configure and manage configuration files than older methods of

scripting (e.g. manually running Shell scripts)[8]. Still, it has its limitations due to the fact that it is vulnerable to the unintended introduction of code smells by developers. [9]

2.2 DevOps

A universal definition of DevOps at the moment is nonexistent, mostly due to the fact that it spans a range of concepts that overlap with each other, making its classification difficult. Yet it can be said that it is a combination of practices, tools, methods, and principles, these include Continuous Software Engineering, that strives for fast, continuous reliable delivery of software[2].

Implementing DevOps typically involves an organizational and cultural shift inside software projects. [10] There is an incentive to break down complex architectures and feature sets into small chunks so that they can be produced and deployed independently. There's also the need to maintain a configuration and build environment that provides constant visibility of what's deployed, with which versions and dependencies. The bridging of the traditionally siloed cultures of development and operations. These guidelines are easier to implement in application and web development, being able to leverage the advantages of cloud computing. An example of this, would be the use of container orchestration systems when implementing a DevOps strategy, such as Kubernetes. Due to its relevance and adoption by DevOps practitioners, Section 2.2.3 provides an introduction to some Kubernetes concepts.

2.2.1 Continuous Software Engineering

Commonly associated with DevOps, Continuous Software Engineering has the aim of integrating various typically separate activities in software engineering into one continuous movement. At its core there are three main methodologies, Continuous Integration, Continuous Delivery and Continuous Deployment [11] [12] [13].

2.2.1.1 Continuous Integration

Continuous Integration, commonly abbreviated to CI, describes a set of practices where developers commit changes to the main branch of a project frequently, these changes then run through automated builds and tests, so that integration problems get detected and resolved accordingly. As CI allows developers to have immediate feedback on their code changes and fix problems earlier in the development cycle, it became a major point of interest in the DevOps movement as smaller and more frequent changes reduce merge and integration issues [13].

2.2.1.2 Continuous Delivery/Deployment

Continuous Delivery, in its two abbreviated forms CD or CDE, involves the use of tests in a staging environment, in order to have production ready software available at all times, yet the deployment process is still manual, involving human action before the product is released. Every change is treated as a potential release candidate to be frequently and rapidly evaluated through

one's continuous delivery pipeline, so that one is always able to deploy and/or release the latest working version, yet one may decide not to [11].

Continuous Deployment, also abbreviated CD, is similar to Continuous Delivery, but it adds automatic deployment to production once staging tests are passed, requiring no human interaction at the end [14]. Figure 2.3 illustrates the actions performed in the CI and CD processes, at the right end of the figure the difference between Continuous Delivery and Deployment is highlighted.

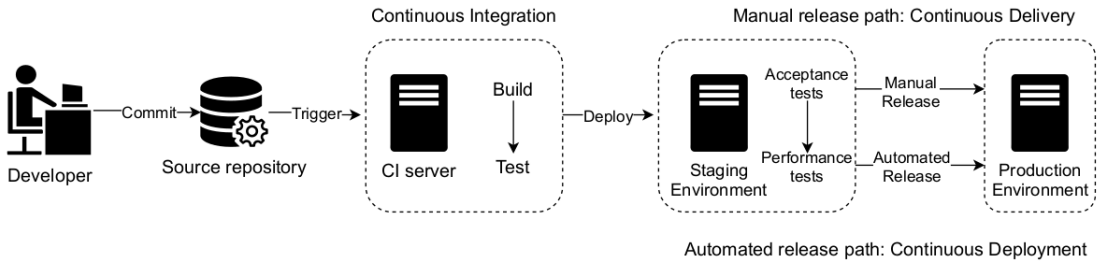

Figure 2.3: CI/CD illustration from [2]

2.2.2 DevOps Tooling

Software associated with DevOps, ranges from tools devoted to configuration management, to continuous integration, deployment of infrastructure through scripts, container runtime environments and container orchestration. In all of these there is an emphasis in automation [10]. An extensive list of DevOps software projects can be found at the Cloud Native Foundation's Landscape [5], and CD Foundation's Landscape [15] websites. It is worth noting that a sizeable proportion of the projects mentioned above are Open Source Software (OSS) solutions.

2.2.2.1 CI/CD Tooling

Here, a brief presentation of some common CI/CD tools will be made.

GitLab and GitHub are software development platforms for project management, version control, CI and source code hosting[16] [17]. The version control component of both is based on Git, a distributed version control system, which gives developers the ability to develop features in parallel by using development branches[18]. Both also provide CI/CD, with build automation, integration testing and staging environments. A GitLab instance can be self-hosted whereas a GitHub one can't.

Jenkins is an open source automation server which can be used to automate tasks related to building, testing, and delivering or deploying software[19]. It is a Java application extensible through plugins. It provides installation options through Docker containers, package managers and standalone files. Its website[19] provides documentation, and walkthroughs on how to deploy it correctly. The pipeline is described in stages which are defined in a script named "Jenkinsfile" written in the Groovy language. This script is interpreted and executed when a build job starts.

It can be edited through the Jenkins GUI or in a text editor. The execution of a typical Jenkins build job is illustrated in figure **??**, in this example the Jenkinsfile script is parsed by a Jenkins master server which then delegates the execution of the tasks to an agent. The agent generates the software artifacts and deploys them to the infrastructure, while the master controls the execution of the stages and the archiving of logs.

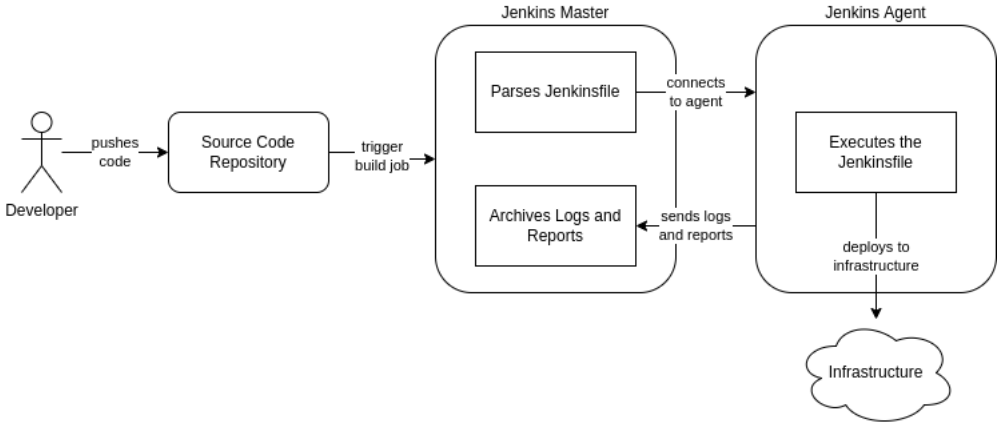

Figure 2.4: Jenkins operation

2.2.2.2 Configuration Management Tooling

Ansible is a Python based IT automation tool for configuring systems, deploying software, and other related IT tasks [20]. Ansible operates in an agentless manner, it does not have dedicated master nodes for management. The tasks to be performed are specified in Playbooks written in the YAML markup language, that will then be parsed and executed by Ansible, which then pushes the changes to its targets. Due to its agentless architecture it makes scalability simpler, at the cost of more centralized control.

2.2.2.3 Containerization Platforms

Docker is a containerization platform for building, deploying, and managing containerized applications [21]. It runs a client-server architecture between its client and daemon, the client is responsible for user interaction and configuration, and the daemon is in charge of the actual building and deployment of container images. Configuration is done by editing docker-compose files and Dockerfiles. Docker can also configure external storage, networking and how isolated a container is from others and its host machine [22].

2.2.2.4 Container Orchestration Tooling

Kubernetes is a container orchestration tool designed to manage and configure distributed container systems (e.g. Microservices). It can act as a load-balancer and storage orchestrator, and has also authentication management, failover and automatic provisioning features. It was

originally designed with scalability in mind by Google, due to their need to deploy large amounts of containers consistently [23].

2.2.3 Kubernetes Concepts

This section will outline some concepts crucial to the operation of a Kubernetes cluster. These typically deal with configuration files written in the YAML format.

2.2.3.1 Pods and Namespaces

Pods are groups of containers with shared storage and network resources, they are the smallest unit of computing one can deploy in Kubernetes. In a way similar to Docker, pods are isolated by Linux namespaces, cgroups and other isolation techniques. They can be thought of as a group of containers with shared namespaces and shared filesystem volumes.

In Kubernetes namespaces, are used to separate resources in a cluster. They provide a scope for certain non cluster-wide components, such as Deployments or Services. They are useful for distributing resources between users or organizing different kinds of software. [24]

2.2.3.2 Deployments and StatefulSets

Deployments describe desired states for pods, in these files users provide details in a declarative form, such as the number of container replicas, open ports, and other metadata for easier querying of information and management. The Deployment Controller is tasked with carrying out the changes that lead to the desired final state. [24]

StatefulSets are similar to Deployments in that they also describe a desired state, but they provide more features for handling stateful use-cases. Each pod in a StatefulSet has a unique identifier, and can be deployed in an ordered manner. It is recommended for use-cases that require unique network identifiers, persistent storage, and ordered updates [24].

2.2.3.3 PersistentVolumes and PersistentVolumeClaims

Persistent storage in Kubernetes is handled using PersistentVolumes and PersistentVolumeClaims. When a Pod is created, by default, no persistent storage is allocated, this means that if a Pod is shut down its data will be lost. To get over this, PersistentVolumes are used for storing the information that would be written to disk. To access a PersistentVolume, a Pod must first be associated with a PersistentVolumeClaim that quantifies the amount of storage requested and its properties (i.e. ReadWrite, ReadOnly...)

To configure PersistentVolumes an appropriate volume provisioner is needed, these exist in the form of plugins. Each plugin may be used for a specific kind of storage protocol, for example NFS or local storage.

2.2.3.4 Secrets and ConfigMaps

ConfigMaps store non-confidential data in key-value pairs, as the name implies, they are commonly used to supply configuration files to the running pods. They are mounted in a way similar to storage volumes.

When there is the need for storing confidential information, for example a private SSH key, it is best not to store it inside a ConfigMap or container image, especially if that image is public. This issue can be circumvented by using a Secret, they behave similarly to a ConfigMap, but they are designed for holding confidential data, it is recommended that they are stored encrypted, as the default is non-encrypted.

2.2.4 Kubernetes Distributions

Kubernetes distributions provide a platform by modifying and reorganizing the main components of standard Kubernetes. They aim to simplify the configuration, startup, and maintenance of low-resource clusters. [25] They aggregate Kubernetes dependencies into a single package for a simpler setup. These distributions, due to how they simplify the configuration of clusters, become opinionated, as in, they reflect certain preferences of the developers of these products in the configurations they choose. Examples of this kind of software are K3s developed by SUSE or MicroK8s developed by Canonical, both backed by enterprises connected to the cloud computing and Linux development realms.

2.3 Security and Testing

As applying DevOps methods to security techniques implies moving security checks to earlier in the development stage, application testing becomes a crucial phase in the security process.

2.3.1 Continuous Security

Continuous Security is an approach for bringing security to DevOps by having security teams adopt DevOps techniques instead of traditional practices more concerned with infrastructure. This way, the focus shifts to the continuous improvement of a product, rather than on posterior fixes [26].

Continuous Security has three main components:

1. Test driven security (TDS): Its goal is to define, implement, and test security controls. It provides security by implementing basic controls and consistently testing those controls for accuracy. Tests should be established for application vulnerabilities, such as those in the Open Web Application Security Project (OWASP) top-10 list [27], infrastructure misconfigurations, for example in SSH servers, VPNs or administration panels and for the CI/CD pipelines themselves.

2. Monitoring and responding to attacks: In order to prepare for a security incident logging, intrusion detection and incident response are critical.

3. Assessing risks and maturing security: Beyond the purely technical aspects of implementing security, risk management and the implementation of a security program play a major role in the overall security of an application or organization.

2.3.2 Vulnerability classification

According to [28], a vulnerability can be described as the existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event compromising the security of the computer system, network, application, or protocol involved. These, when exploited, potentially cause an adverse impact on the system on which it was present.

Security scanning tools identify and classify vulnerabilities according to one or more standards for reference and categorization.

2.3.2.1 CWE and CVE

For the classification of weaknesses, CWE [29] is a list that is widely used. A weakness in this case means a flaw, a fault, a bug, or other errors that can result in systems, networks, or hardware being vulnerable to attack. Each weakness in the CWE list receives a numbered entry, which then can be used to consult the list. This list is a community effort, governed by the Mitre corporation.

Related to the CWE list, there is also the CVE list. CVE entries are specific vulnerabilities found in an implementation, as opposed to CWE entries which are not related to specific software or hardware projects. These can be described in terms of CWE entries in order to classify the vulnerability type. [30] CVEs are also evaluated in terms of severity, by way of CVSS scores. CVEs are given a score from 0 to 10, with a higher CVSS number representing a higher severity vulnerability.

2.3.2.2 OWASP Top Ten

Another classification system is the OWASP Top Ten [27]. It classifies weaknesses similarly to the CWE list, but it organizes them in ten different categories which are a representation of the ten most critical security risks affecting web applications. These are ordered in ascending order of severity. OWASP entries are broad definitions, with this in mind they can be described in terms of CWEs which offer more granular classification. The latest update to the list was released in September 2021, and has the following categories:

- A01:2021-Broken Access Control: details failures of policy enforcement such that users act outside their intended permissions. Includes Path Traversal and Cross-Site Request Forgery.

- A02:2021-Cryptographic Failures: focuses on problems related to cryptography or the lack of it.

- A03:2021-Injection: is concerned with user-supplied data not being validated, filtered, or sanitized by the application. Includes Cross-site Scripting and SQL injection.
- A04:2021-Insecure Design: focuses on risks related to design and architectural flaws.
- A05:2021-Security Misconfiguration: deals with insecure configurations across an application's stack. Includes XML external entity (XXE).
- A06:2021-Vulnerable and Outdated Components: usage of vulnerable components, can be expressed using CVEs.
- A07:2021-Identification and Authentication Failures: related to the confirmation of the user's identity, authentication, and session management.
- A08:2021-Software and Data Integrity Failures: highlights the lack of Integrity verification across software updates critical data and CI/CD pipelines.
- A09:2021-Security Logging and Monitoring Failures: relates to failures in the detection, escalation, and response to active breaches.
- A10:2021-Server-Side Request Forgery: deals with vulnerable applications not validating user-supplied URLs when fetching resources.

2.3.3 Application Testing

2.3.3.1 Static Code Analysis

Static analysis tools examine the text of a program, without executing it. They can examine either a program's source code or a compiled form of the program, although decoding the latter can be difficult. As it has access to the source code and program binaries it is considered a white-box technique. Static analysis tools are faster than manual audits, and they encapsulate security knowledge in a way that doesn't require the tool operator to have the same level of security expertise as a human auditor. Static analysis tools look for a fixed set of patterns, or rules, in the code, if a rule hasn't been written yet to find a particular problem, the tool will never find that problem [31]. Static Application Security Testing (SAST), can be applied early in a CI/CD pipeline after a build stage has been passed, but it can slow down a deployment by flagging false positives that need to be manually reviewed, or by taking too long to scan incremental changes [2].

2.3.3.2 Dynamic Application Testing

In contrast to static analysis, dynamic analysis tests a program while it is running. Dynamic Application Security Testing (DAST), tries to determine how a running application would respond to malicious requests. As it does not have access to the source code or binaries it is considered a black-box technique. These requests are crafted by either an auditor or an automatic scanner, and then sent to the running application[32]. However, this implies that the application must be built,

installed and configured, which only happens in the latter part of the CI/CD pipeline, this coupled with the fact that the automatic tools used for these tasks also need configurations specific to the application being tested, this makes this method costly in terms of time [2].

2.3.3.3 Interactive Application Security Testing

Interactive analysis relies on the monitoring of the code execution in an application. Like SAST 2.3.3.1, it is a white-box technique, but like 2.3.3.2 it requires execution, thus combining properties of both techniques. Generally IAST tools generate fewer false positives than other options, and are easy to correlate with SAST results. Unlike both SAST and DAST, they need to be installed in the server running the targeted application, it also needs some degree of integration with the application. [33]

2.4 DevSecOps

DevSecOps focuses on bringing security to software projects in a way that conforms to the DevOps methodology, thus implementing DevSecOps poses many challenges that need to be addressed, such as the slow pace at which vulnerability assessment programs detect flaws and the inability to completely automate some security processes which limit the speed at which deployments can be made. Developers lacking security skills and complex infrastructure are also contributing factors to the difficulty of an implementation. There are multiple methods for testing and hardening a DevOps pipeline these include Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), Intrusion Detection/Prevention Systems (IDS/IPS), cryptographically secure connections and an effective logging policy. [26] [32] [34].

2.4.1 SAST Tooling

SAST tools, have the capability to not only audit the code commited by the developers to their project's source code repository, but also the third party libraries needed to build the project, if they are open source. These kinds of tasks can be accomplished by programs like Sonarqube, BlackDuck, Snyk and LGTM. These services can be used on-premises or in SaaS subscriptions.

2.4.2 DAST Tooling

There are various options for DAST tools, they range from programs specifically targeted at web applications, like Arachni and OWASP ZAP, to more full-featured suites with support for more protocols and use-cases such as GVM/OpenVA, and Nessus.

There are different DAST methodologies that can be employed. Some tests are based on webcrawlers that after discovering all possible URLs start attacking pages with malicious requests, others are crafted using browser automation tools, such as Selenium, to replicate the behaviour of a typical user of an application, and proxying their traffic through an attack proxy. The most

common of these two approaches is the first one. [32] The results are obtained by analyzing the responses of the application's user interface to the malicious requests.

2.4.2.1 OWASP ZAP scan

OWASP ZAP can be used to perform DAST scans on web applications or as an HTTP proxy. It can perform pre-packaged scans and user-scripted scans. The pre-packaged scans follow a conventional structure. It consists of three stages, first a web crawler proceeds to enumerate all the URLs it finds, then an active scan sends malicious requests to the URLs found earlier, in the final stage a report is produced with the findings from the earlier phases. User-scripted scans can follow a different structure and make use of other plugins, they can also define hooks to call other scripts, they are useful if some kind of authentication is needed.

2.4.3 Runtime Security Tooling

Other tools can monitor the containers runtime during execution, for example tools that rely on eBPF [35], that use system calls to monitor a system by parsing Linux system calls from the kernel at runtime, asserting the returned stream against defined policies and then alerting if a rule has been violated.

2.5 Conclusion

As concerns with norm compliance and security grow, the need for reliable testing of software has increased. This has led to more interest being directed to DevSecOps. Like DevOps, DevSecOps' definition can be ambiguous, literature has been published with the intent of defining this new term [9] [36] [37], explaining the concepts related to an organization's culture and what particular terms mean, but these don't explain the methods used to implement DevSecOps. Others have looked into how organizations struggle with adoption [2], and how to overcome that.

However, literature describing an implementation and its performance is not as common. There have been attempts at this [32], in which DAST and SAST methods were studied in conjunction with GitLab, not including logging, or the design and implementation of the pipeline itself. The performance of tools used in security testing has been documented [33], but its integration in the SDLC was not the main focus. This dissertation will aim to fill the gap in this category by presenting a more complete review of a pipeline's architecture choices in addition to the performance comparison of DevSecOps tools.

Chapter 3

Solution Architecture and CI/CD Software

Over the course of this chapter, the details pertaining to the design and implementation of the DevSecOps pipeline developed during the elaboration of this dissertation, will be presented.

There is no shortage of tools in the DevOps space as we can see in section 2.2.2, as well as cloud provider solutions dedicated to this field, this in turn, makes it so that there are a myriad of ways of crafting a pipeline architecture. The criteria for the selection of the tools and methods presented in this chapter was mostly based on market adoption, whether it is open-source software and not being a complete proprietary solution, e.g. Azure or AWS DevOps solutions.

3.1 Infrastructure and Cloud Providers

3.1.1 Private Cloud, On-Premises and PaaS

As this implementation prioritises open-source solutions that can be deployed on-premises or in a private cloud, cloud providers that offer PaaS (Platform-as-a-Service) solutions, such as AWS, Google Cloud and others, were excluded in favor of providers more in-line with this approach, in this case DigitalOcean and Linode were chosen as the cloud providers.

In contrast to PaaS providers, these do not offer complete proprietary DevOps/DevSecOps solutions. Provisioning computational power, storage, and network resources is the main focus, making the implementation of other features up to the customer. The lack of complex pre-made solutions, consequently brings the monthly cost of these services down, it also helps in keeping a project platform independent.

3.1.2 Cloud Provider Features

Cloud block storage can be provisioned automatically when creating VM instances, by sending requests to the provider's API, or by requesting it through the user interface. This allows for

scalability in case, the need for storage becomes larger. Cloud providers also offer backup services, in case recovery to a previous state is needed, using custom virtual machine images.

Both Linode and DigitalOcean offer managed Kubernetes solutions, these services abstract the Kubernetes control plane, giving the user only worker nodes, this may prove useful in conditions in which the client does not desire the burden of completely administrating a cluster. In this case, these services were not used due to the fact that they limit access to the machines running the workloads, for example not allowing SSH access to the nodes.

3.2 Cluster Configuration

For this setup, three virtual machines were provisioned, each one with access 2 virtual CPU cores and 4 GB of RAM. All virtual machines were running the microK8s Kubernetes distribution, forming a three node cluster. Each node in the cluster functions as both a worker node and control plane node. With a replicated control plane over the three nodes, there is fault-tolerance in case one or two nodes crash, or stop responding correctly to liveness probes. These specifications were chosen by taking into account the workload each virtual machine would be submitted to, in addition to the benefits to fault-tolerance that running three nodes, instead of just two, brings. The nodes share a LAN in which they could deploy applications without them being exposed to the rest of the Web. This aspect is relevant to the testing of an application that could potentially have vulnerabilities.

The operation of microK8s, created an increase in the processor load and RAM usage of each VM similar to that found on [25]. A minimum of 540 MB of free RAM is stated in the microK8 documentation, for normal operation but a regular use-case commonly exceeds this limit. In smaller deployments with limited resources, such as this one, Kubernetes has an impact on the workload capacity of the virtual machines. A managed Kubernetes service that offloads the control plane, can reduce this impact, but at the cost of no access to the underlying VMs.

Cluster administration was performed using the SSH protocol either through the VMs shell in interactive mode, shell scripts or Ansible Playbooks. Authentication was done through password-less SSH, using SSH key pairs for authenticating hosts who would attempt to log in.

3.2.1 Storage

Each one of the machines had access to 80 GB of block storage locally. Distributed storage for the cluster was provided by an NFS server configured in one of the virtual machines. This method centralized storage in a single point, instead of having multiple Kubernetes volumes spread across the three VMs. It makes management simpler, and simplifies the process of switching pods between nodes in the cluster, because permanent storage does not have to move between machines.

An NFS storage provisioner was configured in the cluster, it is deployed as a pod in Kubernetes and enables the cluster to allocate disk space on-demand when volume claims are issued. This provisioner creates and manages the NFS directories to be mounted by the pods, it implements

the NFS protocol for the pods. This way a pod does not have the need for installing NFS tools to mount NFS directories.

Three pods inside the cluster need persistency, the SonarQube server, the SonarQube database, and the Jenkins master. These need to store state information for normal operation and storage of logs. The other components employed during the build process don't need this property, as their state information is not relevant outside the build process, and their outputs are exported to permanent storage in the Jenkins master volume. Thus, when a build job finishes, the contents of the pods are lost.

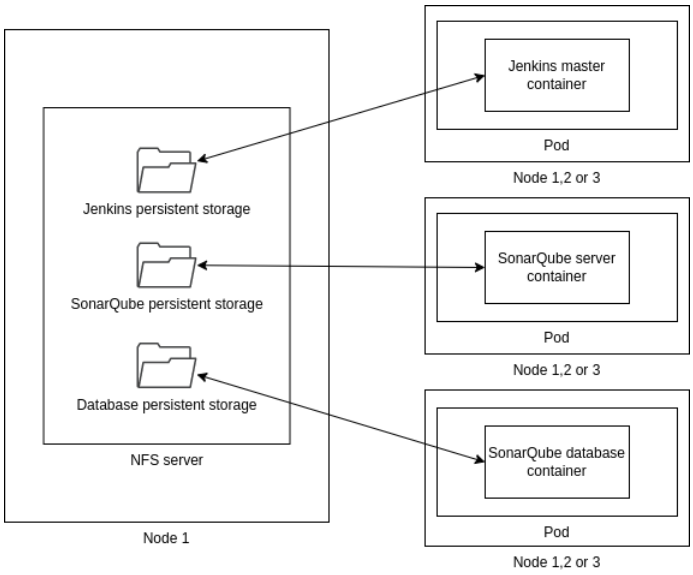

Figure 3.1: NFS server and container mounts

Daily backups to cloud storage were performed using rclone. The cron daemon was used to initiate a backup script daily, that would archive and compress the NFS server's data and upload it to Google Drive. This strategy was chosen due to its ease of implementation, its security and cost efficiency is lackluster when compared to production grade solutions.

3.3 CI/CD software

3.3.1 Jenkins

Due to its widespread adoption and Open-Source nature, the CI/CD tool chosen to be tested was Jenkins. This implementation runs a Jenkins container image in the cluster supplied by the Jenkins developers, available for download at Docker Hub. The version of this container image was the Jenkins LTS (long term support) release version number 2.346.2.

The repositories used in the build process were hosted on GitHub, from which the Jenkins server would pull the source code. Both Jenkins and GitHub provide interoperability, as GitHub provides authentication measures by generating tokens to be used with its API, which then permit

as GitHub to use Webhooks to notify the Jenkins server of a commit, and the Jenkins server to notify GitHub of whether a build job was successful or not.

The container images generated after each build were uploaded to Docker Hub where a repository was previously setup, these are then stored for future deployment.

3.3.2 Kubernetes Integration

For added security, Jenkins build jobs can be delegated to agents, this way the master node does not perform the actual build tasks, and only serves administration purposes. Jenkins when paired with Kubernetes and its container orchestration capabilities can create agents on demand, this integration can be achieved by using the Jenkins Kubernetes plugin, which can be fetched from the Jenkins plugin page. These agents run the Jenkins agent container image, they are connected to the master through an encrypted session on port 50000 using the Java Network Launch Protocol (JNLP) protocol.

The agents when running on Kubernetes, can make use of more than just the Jenkins agent image container, if other functionalities are needed, one can specify a Kubernetes pod declaration in the Jenkinsfile or the Cloud configuration page. This way, features not available to the standard image can be implemented, such as a container running a SAST scan or a container with tools for use in deployment tasks. This is achieved by having the agent interact with the Kubernetes API server, to get access to the other containers when needed.
The agent image can also be customized, as the Dockerfile of this image is Open-Source under the MIT license.

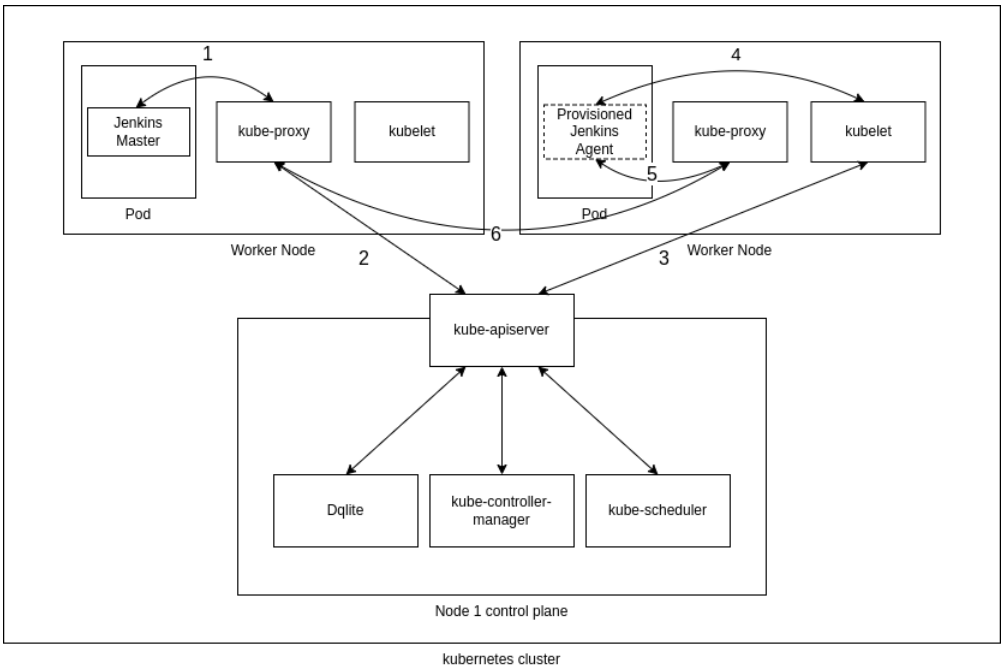

Figure 3.2: Jenkins and Kubernetes interaction

3.3.3 Project compilation and Maven

The application chosen for testing is a Java application, and it uses Apache's Maven for build automation and project management. This tool reads XML files (pom.xml) containing the information that is necessary for the compilation of application's binary, and then executes it. It is through Maven plugins that SonarQube and OWASP's dependency-check interact with the SonarQube server, these are also defined in the pom.xml files.

The remaining pipeline structure is not dependent on Maven or Java, and it can be repurposed for the use of other programming languages or project management tools.

3.3.4 Container image compilation

When compiling Docker container images, access to the Docker daemon is normally needed, which in turn requires root privileges. In Kubernetes this could be implemented by using the "Docker in Docker" approach. By having a container running the Docker daemon while connected to the host system's Docker daemon socket. This ends up giving root access to the container which is not something desirable from a security perspective.

To tackle this problem, third party tools that can compile container images without root access in user-space, have been developed. The pipeline that was put in place makes use of Kaniko, an Open-Source tool maintained by Google, to compile and publish the container images to the registry (Docker Hub). In this use-case Kaniko needs access to Docker Hub credentials, these are passed to the container as a Kubernetes secret volume.

3.3.5 Pod template specification

The agent's pods were specified using a Kubernetes deployment file in YAML format. This file was stored inside the Jenkins server, but it could be written directly in the Jenkinsfile. This pod specification was composed of four containers, one container is the agent responsible for the communication with the master, execution of pipeline steps and running Maven for compilation and execution of the SonarQube, dependency-check plugins, another is the OWASP ZAP container that performs a WAST scan of the target application, and the other two were responsible for executing the Snyk analysis and communication with Snyk's servers. The download of these container images before the start of a build job requires 1549,5 MB of disk space, 250,6 MB for the Snyk Maven image, 301,1 for the Snyk Docker image, and 997,8 MB for the OWASP ZAP image. The time impact of this was mitigated by storing the container images for future builds. It can be further reduced by using Snyk's lighter container images, these do not include build tools. The Snyk Alpine image uses 32 MB of disk space, by substituting the other Snyk images by this one, the total disk space necessary becomes 1061,8 MB.

The YAML file can also be expanded to accommodate new container and volume declarations, if use-cases were to be changed.

```yaml
apiVersion: "v1"
kind: "Pod"
metadata:
  annotations: {}
  labels:
    jenkins: "slave"
    jenkins/label-digest: "eb8b520873bb6b61c45c0696d94d47e6c5e8bdf5" # label hash
    jenkins/label: "kubeagent"  # pod template name
  name: "kubeagent-p72p2"       # pod name with random identifier
  namespace: "jenkins"
spec:
  containers:
    image: "jrolaubi/jenkins-agent-ansible"
    imagePullPolicy: "Always"
    name: "jnlp"
    resources:
      limits: {}
      requests: {}
    tty: true

```

Figure 3.3: YAML Pod template of the Jenkins agent

3.3.6 Pipeline stages

The Jenkinsfile stored in the source code repository of the project is interpreted and executed by the agent. The agent executes the stages defined in the Jenkinsfile in the order shown in 3.4.

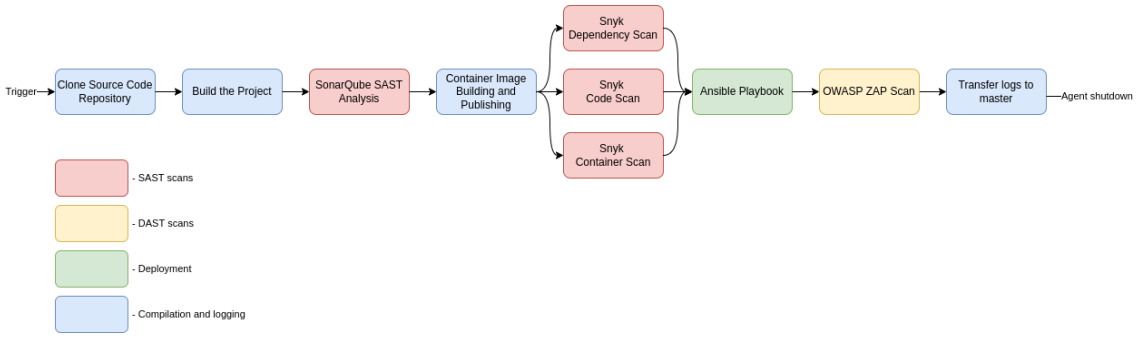

Figure 3.4: Pipeline diagram

The order in which the stages are performed is dependent on certain conditions. As can be seen in 3.4, the testing stages can only start after the project is compiled, this is due to the fact that the project is written in Java and SonarQube needs the .class files to be present, these are generated during compilation, other programming languages may not have this restriction. It can also be seen that the Snyk analysis only happens after the image has been updated to the registry, this has to do with the fact that the Snyk Container scan needs to connect to a registry first. In this implementation, SonarQube and dependency-check were executed in a synchronous manner, while Snyk's components were parallelized using a parallel stage defined in the Jenkinsfile. The OWASP ZAP scan stage can only be executed after a successful deployment to the testing environment, thus

```
stages {
    stage('Example stage') {
        steps {
            sh 'echo "Test"'
        }
    }
}
```

Figure 3.5: Example pipeline stage

it happens after the Ansible Playbook has finished deploying the application to the cluster. This deployment is done by using Ansible's Kubernetes module to interact with the cluster. To prevent IP address disclosure the Ansible hosts file is passed to the containers as a secret, and is not stored in the source code repository. The same is done with the SSH keys, used to authenticate the agents against the cluster's hosts when they need to perform changes to the cluster's deployments.

Further parallelization of the testing workload is possible, as those stages could be executed simultaneously, however for easier measurement of the time taken by each stage the current configuration was used. Apart from the essential compilation and deployment stages, stages can be removed if needed. Additional stages can be added by adding a pipeline stage block to the script.

When executed, the pipeline script follows an algorithm similar to the one shown in 3.6.

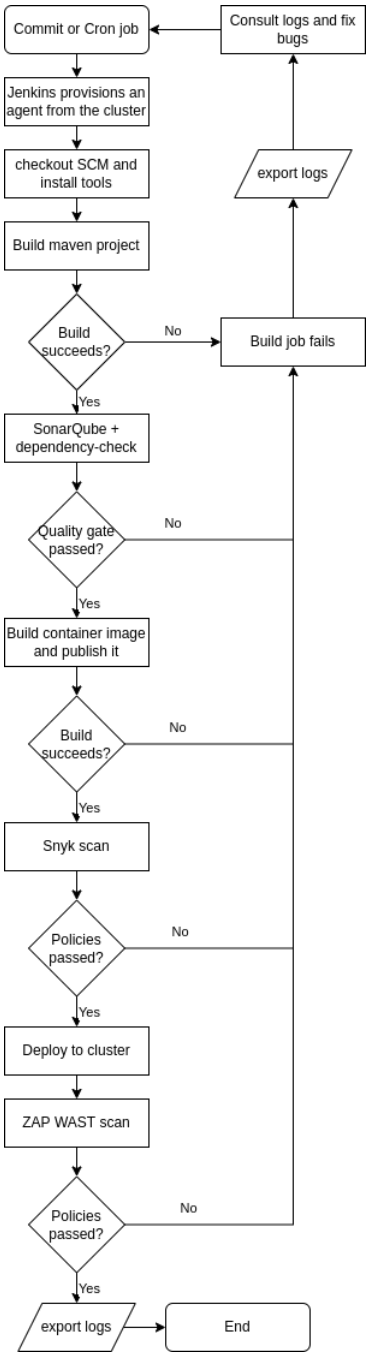

Figure 3.6: Pipeline flowchart

3.4 DevSecOps Frameworks

DevSecOps, being an area that encompasses such a large amount of different practices, involves a wide range of procedures to create a solid implementation strategy. Recent frameworks/-models are being proposed to aid in the validation of DevSecOps procedures. A selection of these were used to evaluate the implementation presented in this document. Some measures proposed are features that are suited for future work in this present implementation.

The OWASP DevSecOPs Maturity Model (DSOMM) defines various guidelines similar to other norms like the ISO's ISO-27001 and OWASP's SAMM2. It rates implementations in four levels of maturity, these levels are then used to evaluate five categories which then are split over eighteen subcategories. The five main categories are "Build and Deployment", "Culture and Organization", "Implementation", "Test and Verification", and "Information Gathering". A visual representation of the model can be generated from the project's webpage.

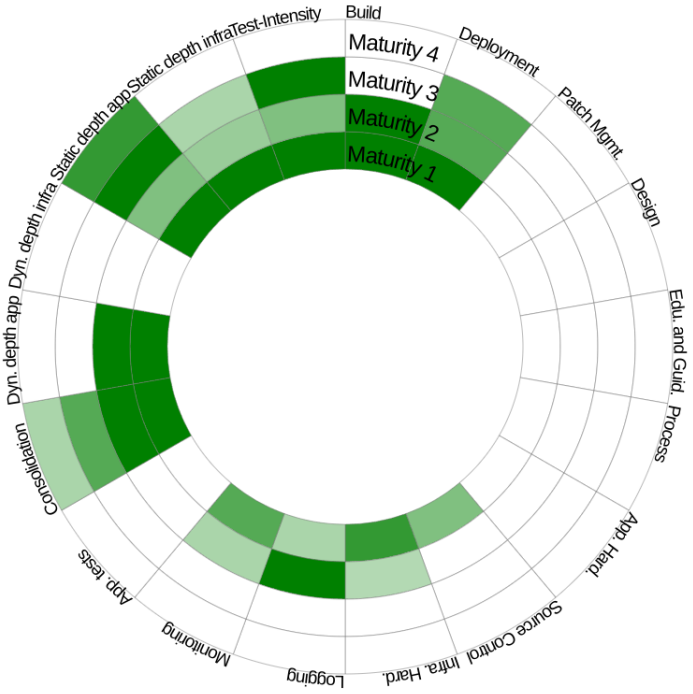

Figure 3.7: Visual representation of the model

As can be observed in 3.7, the work that was documented does not cover part of the categories described in the model. A full-fledged DevSecOps methodology does not rely exclusively on the CI/CD process and application testing. It requires many hardening techniques that are out of the scope of this document that would be crucial in a production implementation.

3.4.1 Build and Deployment stage

In the “Build and Deployment” category, most of the procedures were implemented, however the practices related to patch management were not, as defining a patch policy was outside the scope of this document. In case one was to be defined, practices such as nightly builds, automated pull requests, and container image lifetime limitation could be considered.

The “Build” subcategory achieved important steps, like a defined build process or the use of virtualized environments, having only missed in not signing the code commits or artifacts produced by the pipeline. These two measures would help protect the integrity of the artifacts produced, and make manipulation of the source code more difficult.

The “Deployment” subcategory was also followed to almost completion, with the exception being no downtime updates, a defined image decommissioning policy, feature toggles and Blue/-Green deployments. These were not performed, in part because they would require a larger scale project, as no downtime updates and Blue/Green deployments would need a production environment that would require more resources.

3.4.2 Implementation stage

This stage encompasses infrastructure/application hardening as well as source code control. As the application that was chosen was not modified, application hardening measures were not pursued. Furthermore, WebGoat being an intentionally vulnerable application does not conform to various standards of application security. Source control protection was also not a necessity as the risk of unauthorized edits to the source code was not a concern.

Infrastructure hardening is a major concern that was only partially implemented. Some of the more important measures include encryption at transit, hardened AAA (e.g. role based access control, security accounts, 2FA), IaC. Encryption at transit either internally or at the edge of the network was not implemented, this specific counter-measure is especially important against MITM (Man in the middle) and sniffing attacks.
The AAA features used were mostly the defaults, without much hardening, this is a bigger concern in an organization that needs to manage users on a least-privilege principle.
Even though Kubernetes and containerization implement some aspects of infrastructure as code, the virtual machine images used in the process were not provisioned in an IaC context, with custom configuration done in the interactive command line interface.

3.4.3 Test and Verification stage

In this stage, tests related to application design and some to infrastructure were not explored, as integration tests and network related tests were not performed.

Most entries related to SAST techniques at the application level were put into action except for the static analysis of all individual libraries and dependencies and the integration of static analysis at the IDE level, before code is committed. At the infrastructure level there were fewer guidelines followed, as only tests of the virtualized environments, the container image dependencies, and

exposed secrets were done. The cloud configuration was not tested, malware was not monitored, and the VMs were not checked for vulnerabilities periodically.

The use of a VMS (Vulnerability Management System) was not considered for this implementation, even though centralized vulnerability logging is a desirable feature.

3.4.4 Information Gathering stage

Log treatment in this implementation was partially centralized, while the Jenkins master did store individual reports and the build logs in its NFS volume, other services did not have a centralized logging system where important log events were stored.

The metrics that were collected were basic usage metrics obtained either through the Kubernetes API server or the process monitoring tools found in Ubuntu's repositories, such as top or htop. Observability of metrics is another crucial DevSecOps/DevOps topic, that was not the focus of this dissertation. The DSOMM specifies targeted alerting, metric visualization, monitoring of costs, correlation of security events and centralized application logging as important components to a robust DevSecOps model.

3.5 Conclusion

Over the course of this chapter, the pipeline's component choices were explained, and its structure was delineated. An assessment of its coverage of DevSecOps practices was also conducted by employing a framework based on standardized norms. The next chapter will go into further detail regarding its security testing features and will provide an analysis of the results produced by its reports.

Chapter 4

Security Testing Implementation

The present chapter, focuses on the application security testing aspect of the pipeline described in chapter 3. First, the setup of the testing stages is described followed by an analysis of the results produced by these stages, ending with some considerations on the impact of the testing phases on the pipeline's overall performance.

4.1 Target Web Application

For benchmarking purposes OWASP's WebGoat was chosen as the target of the security tests. OWASP [27] maintains a list of purposefully vulnerable open-source web applications, WebGoat is one of the most popular, its source code is rather easy to compile, and most static analysis tools support its programming language, Java. This Web application is divided into various lessons, with each one covering a specific web application security topic. Ten of these are named after the OWASP Top Ten 2.3.2.2 categories, with the addition of five other topics. At the end of the pipeline, this application is deployed to the three node LAN, this way it remains accessible to all the nodes in the cluster, but not to the outside. The version number of the WebGoat instance used for testing purposes was 8.2.3, the latest version as of March 2022.

4.2 Static Application Security Testing

During the build process, after the successful compilation of application's source code, the Static Analysis stage begins. This stage is formed by two steps, first a SonarQube analysis, and then a Snyk analysis. These generate report files in HTML format for better readability, users can access these through the Jenkins UI or the SonarQube server's UI. The SonarQube version number used in the tests was the 8.9.9 version. Snyk versions ranged from 1.954 to 1.984, this was due to Snyk's upload frequency over the month of July 2022.

```
stage('SonarQube analysis') {
    steps {
        script {
            def scannerHome = tool 'sonarscanner';
            withSonarQubeEnv('sonarqube-webgoat') {
                sh '''mvn sonar:sonar \
                -Dsonar.projectKey=webgoat\
                -Dsonar.host.url=${SONAR_HOST_URL}\
                -Dsonar.login=${SONAR_AUTH_TOKEN}'''
            }
        }
    }
}
```

Figure 4.1: SonarQube pipeline stage

4.2.1 SonarQube Parameters

The implementation was done in an on-prem fashion using the oficially supported Helm chart, this implementation allows the user control over the server unlike other SaaS offerings. SonarQube's SonarScanner is primarily concerned with code quality metrics, it will point out common code smells, bugs and vulnerabilities. The SonarQube server can also track these issues over subsequent scans to ensure they are resolved, it can also track various projects.

In this example, a Jenkins pipeline stage is used to start a SonarScanner instance using the Jenkins SonarQube plugin. Authentication information has to be previously configured in the Jenkins server, and the Maven project must also be compiled for this scan to be successful.

SonarQube can qualify the vulnerabilities it encounters in different categories, such as the OWASP Top 10. It also indicates CWE numbers of those vulnerabilities.

There is the option of integrating OWASP's dependency-check as a third-party plugin, this enables the detection of vulnerable dependencies in SonarQube. Other third-party plugins can be used for defining new analysis rules or exporting documents to a specific format for further analysis.

4.2.2 Snyk Parameters

Unlike SonarQube, Snyk does not provide a way to manage a server inside the user's premises, so when a scan occurs, it performs requests to a remote server controlled by the Snyk company, which then responds to the Snyk application.

Snyk has various components in its platform that can perform Static Analysis, in this case the Open-Source, Code, and Container modules were used. These modules distinguish themselves by focusing on specific areas, for example the Open-Source module concerns itself with finding vulnerable Open-Source libraries in the dependencies of the code, the Code module performs more general Static Analysis techniques, and the Container module examines the dependencies of the container images.

The container module specifically distinguishes Snyk from SonarQube, as there is not a similar feature implemented in that product, and it highlights an important attack surface.

Snyk can be integrated in Jenkins in two manners, either using the Jenkins plugin or through a Snyk container image. The first approach, is limited to the Snyk Open-Source module. The second option has access to the all Snyk CLI commands. In order to not leak Snyk credentials, a user must be aware that he shouldn't pass credentials to the commands using Groovy string interpolation (i.e. using double quotes), as this can cause the credentials to appear in the pipeline log.

One can interact with the containers by issuing commands through the Jenkins kubernetes plugin command interface.

```
stage('Snyk analysis') {
    steps {
        catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
            snykSecurity(
            snykInstallation: 'snyk',
            snykTokenId: 'snyk',
            additionalArguments: '--debug --all-projects'
            )
        }
    }
}
```

Figure 4.2: Snyk plugin stage

```
stage('Snyk Code scan') {
    steps {
        container('snyk-docker') {
            catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
                sh '''
                    snyk auth ${SNYK_TOKEN}
                    snyk code test --json \
                    --debug | snyk-to-html -o code-results.html
                    '''
            }
        }
    }
}
```

Figure 4.3: Snyk Code container stage

4.3 Dynamic Application Security Testing

In the penultimate stage of the build process, after the Static Analysis stage finishes, a Dynamic Application test is performed using OWASP ZAP, the version number being used for the tests is 2.9.11. This stage also generates a report file in HTML format for better readability, users can also access these through the Jenkins UI.

4.3.1 OWASP ZAP parameters

OWASP ZAP has various kinds of tests, in a Python script format, that can be used in a CI/CD scenario, these are included in the official container image, and can be invoked by sending commands to the container. Users can also define their own scans by making use of the Automation Framework, in which users can declare a scan structure in YAML format. ZAP can also be configured to interact with scripts for general purposes.

For this purpose the pre-packaged full scan and the Automation Framework were used. These two have three stages, first a standard web crawler which maps all the webpages it encounters, then a second web crawler targeted at modern JavaScript-based Ajax web applications, and last an active scan of the vulnerabilities in each found webpage, these vulnerabilities are then reported and classified by severity.

The pre-packaged scan method can be an effective way to test an application, especially if no authentication is needed, however in WebGoat's case it has its problems. First, WebGoat requires authentication, this forces the user to generate a context file using the Desktop GUI and then exporting it to the container, in order to authenticate properly. Then, some of WebGoat's pages are not accessible by ZAP's crawlers, this is the case for the lesson pages, which are the intentionally vulnerable ones, to overcome this, additional scripts would have to be called during the scans execution

So, because of the added complexity this method would bring, tests with the need of directory enumeration and authentication were performed using the Automation Framework. Which, through its easier to read YAML syntax, gives the user the means to perform these tasks without having to write additional scripts. Tasks are specified as a YAML list, which then enumerates the parameters for each task, in 4.4 an example of an active scan task is shown.

```yaml
- parameters:
    context: "WebGoat"
    user: "testing"
    policy: "Default Policy"
    maxRuleDurationInMins: 0
    maxScanDurationInMins: 0
  policyDefinition:
    defaultStrength: "medium"
    defaultThreshold: "off"
    rules: []
  name: "activeScan"
  type: "activeScan"

```

Figure 4.4: ZAP YAML automation example

To complete the authentication process it was necessary to create a user account prior to testing, this was achieved by running a simple Python script that would send a POST request with the account's details to the registration page, with the credentials specified as environment variables. ZAP would then log in using these credentials passed to it as Jenkins credential environment

variables, and then would do conventional session management with a JWT passed as a cookie.

```
import sys
import requests

url = sys.argv[1]
username = sys.argv[2]
password = sys.argv[3]

req = requests.post(url, data={'username':username,'password':password,'
    matchingPassword':password,'agree':'agree'})
```

Figure 4.5: Registration script

4.4 Results and Discussion

The information discussed in the next subsections was obtained by analyzing the log outputs of the SAST and DAST tools stored in the Jenkins master node, in the form of JSON and HTML formats, and the Jenkins master timestamps for each stage of the pipeline. The time data used was obtained by manually registering the values that the Jenkins master UI had shown.

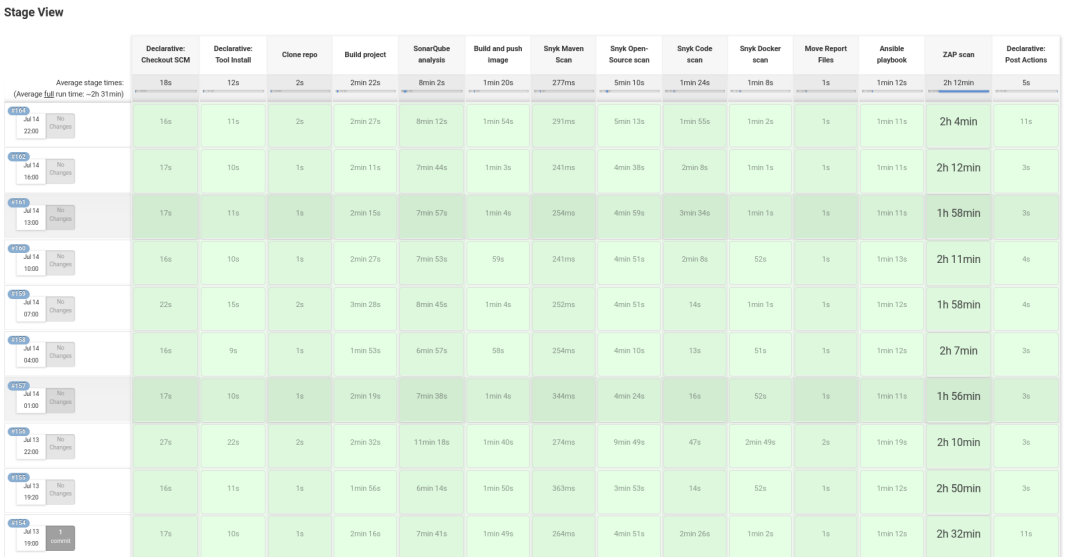

Figure 4.6: Jenkins build job timing

The graphs and tables related to vulnerability data namely CVE and CWE numbers were generated by extracting the values through regular expression pattern matching of the JSON keys, or using Python's JSON parser, and then using Python's matplotlib library for obtaining an image.

Each tool had its JSON key-value pair structure, which meant custom scripts for each tool had to be written. The exceptions to this were Snyk Container and Snyk Open Source which used the same structure.

4.4.1 Impact on Build Times

Both methods do have considerable impacts in build times, however the DAST method is much more time-consuming. Test times in the SAST options did not take more than twenty minutes to complete. While a full dynamic scan of the application would take approximately two and a half hours.

These time differences highlight the different nature and purpose of both tests. As a DAST scan can impact the deployment speed of a project considerably, in a CD scenario this can lead to higher lead time for changes and lower deployment frequency. Both static analysis and WAST also introduced considerable dispersion in the time that a build job would take to complete. This in turn, makes the pipeline less predictable, as it is more difficult to predict when a job will terminate.

$$\mu = 2h29m \qquad \sigma = 16.7m \tag{4.1}$$

job ID	time	deviation
149	2h16m	13m
152	2h16m	13m
154	2h51m	22m
155	3h6m	37m
156	2h38m	9m
157	2h14m	15m
158	2h23m	6m
159	2h18m	11m
160	2h30m	1m
161	2h17m	12m

Table 4.1: Build job duration

This test also evaluates if a service like Snyk's SaaS offering has an advantage over the completely on-prem SonarQube plus dependency-check in terms of performance. The on-prem option needs to do the dependency checking stage and static analysis synchronously (if one wants reporting also inside the SonarQube server), as opposed to Snyk's possibility of parallelism, due to the self-contained nature of each component. Snyk Code also takes less time scanning a project than its counterpart, SonarQube, with the majority of scan time being spent on the dependency inspection phase.

The addition of the test stages has resulted in an increase in the duration of the build job of 94% in Snyk's case and 136% in SonarQube+dependency-check's case. While the WAST analysis of ZAP had an impact of 2439%.

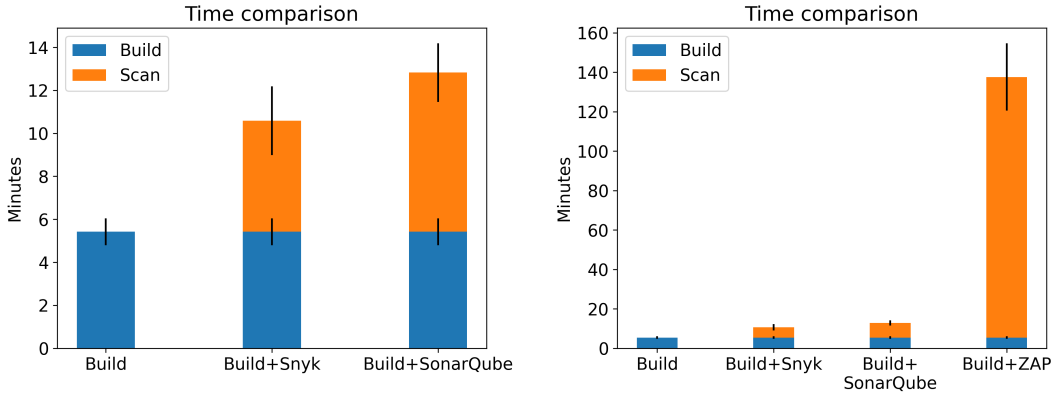

Figure 4.7: Time comparison of SAST and DAST implementations

4.4.2 Vulnerabilities Found

Both kinds of scans reported a considerable number of vulnerabilities, due to outdated dependencies, intentional weaknesses and insecure configurations related to the implementation details. The nature of the vulnerabilities found by each component will be discussed in the following subsections.

4.4.2.1 Snyk, SonarQube and dependency-check

A rough equivalence in functionalities can be established between the components of each solution. SonarQube's static code analysis is similar in function to that of Snyk Code, and dependency-check performs the same role as that of Snyk Open-Source. Meanwhile, Snyk Container's dependency checking at the container image level does not have an equivalent in the SonarQube plus dependency-check configuration. Using this feature, Snyk also checks other binaries shipped in the container image for vulnerable dependencies broadening the scope of the analysis.

The version of WebGoat that was used for running tests was forked from the main repository in the 31st of March. As no git fetch or git pull actions were run, the project's dependencies have become outdated. Snyk Open-Source and dependency-check were able to catch a number of CVEs that were present in the outdated versions of the libraries. In addition, WebGoat makes use of specific out-of-date libraries in some of its lesson pages. Both tools produced reports in a human-readable html format, and JSON. After parsing the JSON version of the report, the number of unique CVEs found and the respective CVSS scores was retrieved. Snyk also flagged three vulnerabilities without a CVE number.

Tool	n CVEs	n unique CVEs	vulnerable dep.	dep. scanned
dependency-check	129	99	27	285
Snyk	21	18	40	269

Table 4.2: Vulnerable dependencies found

Even though dependency-check flags more CVEs than Snyk, it reports less vulnerable dependencies than its counterpart, this is due to the way Snyk counts its dependencies. Snyk divides the WebGoat project into four different modules, this has to do with how the Maven project is structured, because of this a dependency can be counted as vulnerable more than once if it is present in more than one module.

Considering the CVSSv3 scores, dependency-check found more CVEs in all ranges except in the 4.0 to 6.0 range, this is especially relevant in the High to Critical severity range, 7.0 to 8.9 and 9.0 to 10.0 respectively.

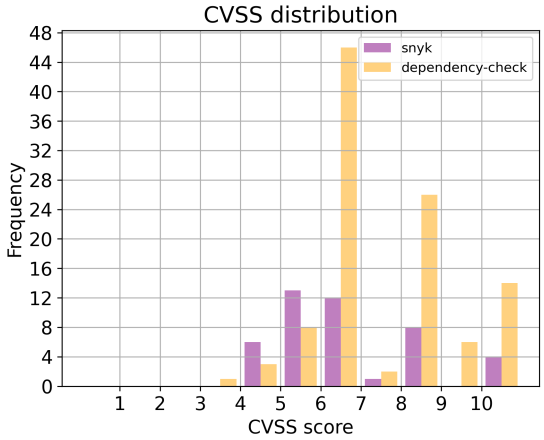

Figure 4.8: CVSSv3 score distribution of Snyk and dependency-check

In the static analysis of code, SonarQube and Snyk, both flag issues by category, CWEs, and severity. After inspecting the output of the reports that were produced, it is possible to conclude that many of the same CWEs were detected, though SonarQube flags more CWEs as it also includes related ones. This last observation makes CWE count not a very good indicator of how many vulnerabilities were encountered, this is due to the nature of CWEs, which are more generic indicators of a weakness than a specific vulnerability. One can also observe that Snyk's categorization of vulnerabilities is more granular. In the following tables 4.4 4.3 the output of the reports are compiled.

At the container image level of analysis, done exclusively using Snyk, a higher number of vulnerabilities were found. From the entries that possessed a disclosure time field, a table with the number of vulnerabilities per year was compiled. 4.5 The report that was used was generated the 4th of July 2022. It can be concluded that the majority of vulnerabilities were from the same year the analysis took place or the previous one. This suggests that a number of vulnerabilities could be mitigated by pulling the new WebGoat version from its repositories, as the new version also uses a more up-to-date base image in its Dockerfile.

Vulnerability	n. Occurrences	Severity	CWEs
Authentication	14	High	CWE-798,CWE-259
CSRF	22	High	CWE-352
SQLi	11	High	CWE-89
XSS	5	High	CWE-79,CWE-1004
DoS	2	Medium	CWE-400
Cryptography	16	Medium	CWE-326,CWE-327,CWE-330,CWE-338,CWE-916
XXE	1	High	CWE-611,CWE-827
Configuration	9	Low	CWE-311,CWE-315,CWE-614,CWE-489,CWE-215
Others	2	Low	CWE-409,CWE-377,CWE-379

Table 4.3: SonarQube SAST table

Vulnerability	n. Occurrences	Severity	CWEs
XSS	10	High	CWE-79
SQLi	18	High	CWE-89
Path Traversal	9	High	CWE-23
JWT Signature Verification Bypass	8	High	CWE-347
Hard-coded Constants	14	High	CWE-547
CSRF	2	High	CWE-352
XXE	2	High	CWE-611
Insecure Deserialization	2	High	CWE-502
Cookie HttpOnly flag	3	Medium	CWE-1004
CRLF Sequences HTTP headers	3	Medium	CWE-113
Hard-coded credentials	8	Medium	CWE-798,CWE-259
Weak Cryptography	1	Medium	CWE-916
Permissive Cross-domain policy	1	Medium	CWE-942
HTTPS without Secure	1	Medium	CWE-614

Table 4.4: Snyk SAST table

Year	n CVEs
2022	113
2021	112
2020	27
2019	25
2018	9
2017	21

Table 4.5: Vulnerable dependencies by year

4.4.2.2 OWASP ZAP

The non-deterministic nature of ZAP's WAST scanning, affects the number of vulnerabilities found across the reports. In this case, a discrepancy between the number of SQL injection, Path Traversal, Remote OS code injection, and XSS vulnerabilities was found. To illustrate this, 4.6 contains the number of SQL injections found over the course of ten build jobs.

$$\mu = 12.4 \qquad \sigma = 5.99 \tag{4.2}$$

job ID	SQLi	deviation
149	8	4.4
152	10	2.4
154	25	12.6
155	23	10.6
156	7	5.4
157	11	1.4
158	10	2.4
159	8	4.4
160	10	2.4
161	13	0.6

Table 4.6: SQL injection per build job

This poses a question of whether some of these vulnerabilities are actually false positives or genuine ones. Some of these vulnerabilities could be unintended ones, not related to the lessons. In fact, some responses WebGoat gives could be interpreted as a vulnerability, even though they are just hints of one, present for educational purposes. Coupled with the black-box nature of DAST, the process of finding the root cause of the warning becomes more difficult. ZAP only lists the URL, parameters, and HTTP response where the perceived vulnerability was found, not highlighting any specific code.

Description	SQL injection may be possible.
URL	http://192.168.128.54:30680/WebGoat/SqlInjection/assignment5a
Method	POST
Parameter	account
Attack	'Smith'
Evidence	unexpected token:
Show / hide Request and Response	

Figure 4.9: Presentation of a SQL injection in a ZAP report

After consulting a list WebGoat's intended vulnerabilities [38] and ZAP's logs, it became apparent that most of the SQL injections, and the Path Traversals, exploits were indeed false positives. Some were presented with no specific evidence. Part of the lessons included in WebGoat contain mitigations that an automated DAST scan would find more difficult to surpass.

Category	n. Ocurrences	Risk level	CWEs
Cross Site Scripting (Reflected)	2	High	CWE-79
Path Traversal	1	High	CWE-22
SQL Injection	8	High	CWE-89
Absence of Anti-CSRF Tokens	112	Medium	CWE-352
Application Error Disclosure	1	Medium	CWE-200
Content Security Policy (CSP) Header Not Set	46	Medium	CWE-693
Format String Error	1	Medium	CWE-134
Parameter Tampering	18	Medium	CWE-472
Vulnerable JS Library	6	Medium	CWE-829
Application Error Disclosure	66	Medium	CWE-200
Cookie No HttpOnly Flag	1	Low	CWE-1004
Cookie without SameSite Attribute	1	Low	CWE-1275
Cross Site Scripting Weakness (JSON Response)	14	Low	CWE-79
Information Disclosure - Debug Error Messages	66	Low	CWE-200
Private IP Disclosure	1	Low	CWE-200
Timestamp Disclosure - Unix	15	Low	CWE-200

Table 4.7: ZAP vulnerabilities and CWEs

Only one of the 110 reports generated by ZAP generated true positive SQL injections, these were classified as Hypersonic SQL injection vulnerabilities in the report. ZAP found 11 of these True Positives in a total of 34 total vulnerabilities. The Precision of this report amounts to 32,3%.

$$Precision = \frac{TP}{TP+FP} * 100 = \frac{11}{11+23} * 100 = 32,3\% \tag{4.3}$$

This highlighted how many of the actual vulnerabilities would be difficult to find using exclusively an automated DAST scan. Increasing the test intensity may reduce the number of false positives encountered, though this will make testing more time intensive.

ZAP supports four attack strength settings:

- Low: Maximum 6 requests
- Medium: Maximum 12 requests
- High: Maximum 24 requests
- Insane: No specified maximum requests, potentially hundreds

As the medium strength policy was used, if the intensity were to be increased for the whole application, then the build job time could potentially double.

Most of the warnings present in 4.7 are related to HTTP responses that highlight some kind of flaw or misconfiguration, like information disclosure or insecure HTTP headers. As ZAP does not have access to the code that may originate these warnings, it will flag all sources of them, resulting in numerous warnings.

4.5 SAST and DAST integration in DevSecOps

Both SAST and DAST, present some integration problems in a CI/CD scenario, with the lead time to change impact being a great concern in these kinds of setups. Organizations that aim to deploy on-demand or several times a day as an SLO (Service Level Objective) are especially susceptible to this factor, like the highest performers mentioned in [39]. Parallelization of the testing workload was only used partially in this specific implementation 3.3.6, further optimization is desirable in order to lower the build and scan times. Part of Snyk's performance edge may be due to this fact, but it also could be that its SaaS offering has better performance than SonarQube's on-premises scanner.

As previously mentioned in 4.4.1, there was a considerable difference in the impact on build times between both techniques, with SAST options completing a full scan of the application approximately twenty times faster in some samples. This suggests that defining a more detailed application testing strategy is desirable if there is the need for faster deployments, as the time needed for a full DAST scan may not be available. DAST test intensity may also be scaled down, either by limiting the number of requests a tool can make or by just testing specific components of an application. Defining different deployment environments with different testing intensities can also help in reducing the time spent on a build job, as builds directed to staging environments might not be put to the same scrutiny as one destined to production workloads.

The precision of both options is also a factor in the process of manual code review that is necessary for the identification of true and false positives. Such process, requires knowledge of the codebase and security notions. This is especially important in DAST as was shown earlier in 4.4.2.2, considering that warnings are given based on URLs and not specific lines of code, due to its black-box nature. The amount of hours dedicated to these kinds of tasks will be higher in DAST scans, and consequently fixing flagged issues will be more expensive to an organization. Both SonarQube and Snyk offer user interfaces that make the process of suppressing known false positives easier, ZAP does the same by specifying alert filters in the automation framework.

Fine-tuning of the scan parameters might also wield better results when applied to DAST. Different components of an application require different intensities in testing. The analysis of simple header messages can be accomplished with low intensity, while other more complex exploits may require more requests.

These measures however, do not ensure a that the DevSecOps methodology being practiced is necessarily a robust one, as they mostly focus on the safety of the artifacts produced by the pipeline and not on the security of the pipeline itself. Some procedures were described when commenting on the implementation, but these are still not sufficient, as was seen when the OWASP DSOMM was discussed 3.

Chapter 5

Conclusion

The work produced in this dissertation documented the design and performance of a DevSecOps pipeline, and how its security testing methods impact the delivery of software in a CI/CD scenario.

The design decisions and implementation of the pipeline documented in chapter 3 were made relying on cloud features, but without recourse to PaaS options. In order to delineate a reusable architecture reliant on open-source options. This chapter also showed how a security testing methodology can be integrated in the CI/CD pipeline, with a higher focus on the CI aspect. An appraisal of the coverage of DevSecOps pratices by the pipeline was also done 3.4, this highlighted the breadth of measures that need to be taken to create a robust system. A study of all procedures was not undertaken, due to the scale of this task, instead the focus was put on the application testing phase of the CI process.

In chapter 4, an assessment of the impact that each scan type had on the duration of the software building process was performed, from which it was concluded that SAST options, while still having a considerable time impact, were the more time efficient testing method. In addition, the results produced by the testing tools showed that the non-deterministic nature of DAST tools made the identification of vulnerabilities more difficult, this, coupled with a low precision when detecting specific categories vulnerabilities and a black-box testing paradigm, makes this testing approach require more hours of manual code review. This contrasts with SAST tools, which can point out specific vulnerable lines of code, making the code review process much faster. The metrics that were collected were also used to compare two SAST solutions, one hosted on-premises while the other used a SaaS model. Both tools outperformed each other in different categories, Snyk had the advantage in test duration and code analysis, while SonarQube/dependency-check solution handled Java dependency checking better.

The work presented in this document has illustrated the advantages and disadvantages that may arise from the implementation of both SAST and DAST testing methods in a CI/CD environment, while also documenting the design decisions of the underlying pipeline system.

5.1 Future Work

Regarding the possibilities of future work, many of the procedures mentioned in 3.4 present avenues for new work to be developed, namely in the infrastructure hardening and information gathering categories. As for the information gathering category, a more efficient logging system, making use of a centralized tool for the interpretation of vulnerabilities such as a VMS (Vulnerability Management System), could be a worthy addition.

Related to the testing of artifacts, some additions that should be studied include:

- IAST described in section 2.3.3.3 is a testing technique that was not employed in this dissertation.
- The efficiency of container registries and container vulnerability/compliance scanning tools.
- Runtime security solutions, such as those based on eBPF.

Further investigation could also make use of other web applications in addition to WebGoat, as this would help in broadening the analysis of vulnerabilities, one specific example is the OWASP Benchmark which provides its own tool for benchmarking scanning tools by identifying the true and false positives in the scan reports.

Appendix A

Jenkinsfile code

```
pipeline {
    triggers{
        githubPush()
        cron('''0 */3 * * *''')
    }
    agent {
        kubernetes {
            label 'kubeagent-webgoat'
        }
    }
    tools {
        maven 'maven'
        jdk 'jdk17'
    }
    environment {
        SERVER_ADDR = '192.168.128.54'
        SCAN_URL_SITE = 'http://192.168.128.54:30680'
        SCAN_URL_YAML = 'http://192.168.128.54:30680/WebGoat'
        SCAN_URL_PYTHON = 'http://192.168.128.54:30680/WebGoat/login'
    }
    stages {
        stage('Clone repo') {
            steps {
                checkout scm
            }
        }
        stage('Build project') {
            steps {
                sh '''sed -i "s/0.0.0.0/${SERVER_ADDR}/" ${WORKSPACE}/docker/start.
    sh'''
                sh '''cat ${WORKSPACE}/docker/start.sh'''
                sh 'mvn clean install -DskipTests'
            }
        }
        stage('SonarQube analysis') {
```

```
35          steps {
36              script {
37                  def scannerHome = tool 'sonarscanner';
38                  withSonarQubeEnv('sonarqube-webgoat') { // If you have
    configured more than one global server connection, you can specify its name
39                      sh '''
40                      mvn org.owasp:dependency-check-maven:7.1.1:aggregate -
    Dformats=JSON,HTML
41                      mkdir ${WORKSPACE}/dependency-check-reports
42                      cp ${WORKSPACE}/target/dependency-check-report.json ${
    WORKSPACE}/dependency-check-reports/
43                      cp ${WORKSPACE}/target/dependency-check-report.html ${
    WORKSPACE}/dependency-check-reports/
44                      mvn sonar:sonar \
45                      -Dsonar.projectKey=webgoat\
46                      -Dsonar.host.url=${SONAR_HOST_URL}\
47                      -Dsonar.login=${SONAR_AUTH_TOKEN}'''
48                  }
49              }
50          }
51      }
52      stage('Build and push image') {
53          environment {
54              PATH        = "/busybox:$PATH"
55              REGISTRY    = 'index.docker.io'
56              REPOSITORY  = 'jrolaubi'
57              IMAGE       = 'webgoat-tese'
58          }
59          steps {
60              script {
61                  container(name: 'kaniko', shell: '/busybox/sh') {
62                  sh '''#!/busybox/sh
63                  /kaniko/executor -f `pwd`/docker/Dockerfile -c `pwd`/docker --
    build-arg webgoat_version=8.2.0-SNAPSHOT --cache=true --destination=${REGISTRY
    }/${REPOSITORY}/${IMAGE}
64                  '''
65                  }
66              }
67          }
68      }
69      stage('Snyk Scan') {
70          failFast true
71              environment {
72                  SNYK_TOKEN = credentials('snyk-api')
73              }
74          parallel {
75              stage('Snyk Open-Source scan') {
76                  steps {
```

```
77                      catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE')
    {
78                          container('snyk-maven') {
79                              sh '''
80                                  snyk auth ${SNYK_TOKEN}
81                                  snyk test --json --json-file-output=maven-
    results.json \
82                                  --debug --all-projects | snyk-to-html -o maven-
    results.html
83                                  '''
84                          }
85                      }
86                  }
87              }
88              stage('Snyk Code scan') {
89                  steps {
90                      container('snyk-code') {
91                          catchError(buildResult: 'SUCCESS', stageResult: '
    FAILURE') {
92                              sh '''
93                                  snyk auth ${SNYK_TOKEN}
94                                  snyk code test --json --json-file-output=code-
    results.json \
95                                  --debug | snyk-to-html -o code-results.html
96                                  '''
97                          }
98                      }
99                  }
100             }
101             stage('Snyk Docker scan') {
102                 steps {
103                     container('snyk-docker') {
104                         catchError(buildResult: 'SUCCESS', stageResult: '
    FAILURE') {
105                             sh '''
106                                 snyk auth ${SNYK_TOKEN}
107                                 snyk container test --json --json-file-output=
    docker-results.json \
108                                 jrolaubi/webgoat-tese \
109                                 --file=`pwd`/docker/Dockerfile | snyk-to-html -
    o docker-results.html
110                                 '''
111                         }
112                     }
113                 }
114             }
115         }
116     }
117     stage('Move Report Files') {
```

```
118             steps {
119                 container('snyk-maven') {
120                     sh '''
121                         mkdir ${WORKSPACE}/snyk-reports
122                         mv ${WORKSPACE}/maven-results.* ${WORKSPACE}/snyk-reports
123                         mv ${WORKSPACE}/code-results.* ${WORKSPACE}/snyk-reports
124                         mv ${WORKSPACE}/docker-results.* ${WORKSPACE}/snyk-reports
125                         '''
126                 }
127             }
128         }
129         stage('Ansible playbook') {
130             steps {
131                 ansiblePlaybook(
132                 playbook: '${WORKSPACE}/ansible/deploy.yml',
133                 inventory: '/home/jenkins/ansible/inventory')
134             }
135         }
136         stage('ZAP scan') {
137             environment {
138                 WEBGOAT_CREDENTIALS = credentials('webgoat')
139             }
140             steps {
141                 timeout(time: 30, unit: 'SECONDS') {
142                     waitUntil {
143                         script {
144                             try {
145                                 def response = httpRequest "${SCAN_URL_YAML}/
    registration"
146                                 return (response.status == 200)
147                             }
148                             catch (exception) {
149                                  return false
150                             }
151                         }
152                     }
153                 }
154                 script {
155                     container(name: 'zap', shell: '/bin/sh') {
156                         sh '''#!/bin/sh
157                         export PATH=/zap:$PATH
158                         mkdir /zap/wrk
159                         mv ${WORKSPACE}/zap/zap.yml /zap/zap.yml
160                         mv ${WORKSPACE}/zap/createAccount.py /zap/createAccount.py
161                         sed -i "s,REPLACE_SITE,${SCAN_URL_SITE},," /zap/zap.yml
162                         sed -i "s,REPLACE,${SCAN_URL_YAML},," /zap/zap.yml
163                         python3 /zap/createAccount.py ${SCAN_URL_YAML}/register.mvc
     ${WEBGOAT_CREDENTIALS_USR} ${WEBGOAT_CREDENTIALS_PSW}
164                         /zap/zap.sh -cmd -autorun zap.yml
```

```
                        cp -r /zap/wrk ${WORKSPACE}/zap-report
                        '''
                    }
                }
            }
        }
    }
    post {
        always {
            // publish html
            publishHTML target: [
                allowMissing: false,
                alwaysLinkToLastBuild: false,
                keepAll: true,
                reportDir: './zap-report',
                reportFiles: 'index.html',
                reportName: 'OWASP Zed Attack Proxy'
            ]
            publishHTML target: [
                allowMissing: false,
                alwaysLinkToLastBuild: false,
                keepAll: true,
                reportDir: './dependency-check-reports',
                reportFiles: 'dependency-check-report.html',
                reportName: 'dependency-check report'
            ]
            publishHTML target: [
                allowMissing: false,
                alwaysLinkToLastBuild: false,
                keepAll: true,
                reportDir: './snyk-reports',
                reportFiles: 'docker-results.html',
                reportName: 'Snyk Docker results'
            ]
            publishHTML target: [
                allowMissing: false,
                alwaysLinkToLastBuild: false,
                keepAll: true,
                reportDir: './snyk-reports',
                reportFiles: 'code-results.html',
                reportName: 'Snyk Code results'
            ]
            publishHTML target: [
                allowMissing: false,
                alwaysLinkToLastBuild: false,
                keepAll: true,
                reportDir: './snyk-reports',
                reportFiles: 'maven-results.html',
                reportName: 'Snyk Maven results'
```

```
214                     ]
215                 }
216             }
217 }
```

References

[1] Claus Pahl. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31, May 2015. Conference Name: IEEE Cloud Computing. doi:10.1109/MCC.2015.51.

[2] Roshan N. Rajapakse, Mansooreh Zahedi, M. Ali Babar, and Haifeng Shen. Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology*, 141:106700, January 2022. URL: https://www.sciencedirect.com/science/article/pii/S0950584921001543, doi:10.1016/j.infsof.2021.106700.

[3] Omer Rana. The Costs of Cloud Migration. *IEEE Cloud Computing*, 1(1):62–65, May 2014. Conference Name: IEEE Cloud Computing. doi:10.1109/MCC.2014.24.

[4] Peter M. Mell and Timothy Grance. The NIST Definition of Cloud Computing. September 2011. Last Modified: 2018-11-10T10:11-05:00. URL: https://www.nist.gov/publications/nist-definition-cloud-computing.

[5] CNCF. Cloud Native Computing Foundation. URL: https://www.cncf.io/.

[6] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, May 2018. Conference Name: IEEE Software. doi:10.1109/MS.2018.2141030.

[7] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. DevOps: Introducing Infrastructure-as-Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, Buenos Aires, May 2017. IEEE. URL: http://ieeexplore.ieee.org/document/7965401/, doi:10.1109/ICSE-C.2017.162.

[8] Akond Rahman and Laurie Williams. Different Kind of Smells: Security Smells in Infrastructure as Code Scripts. *IEEE Security Privacy*, 19(3):33–41, May 2021. Conference Name: IEEE Security Privacy. doi:10.1109/MSEC.2021.3065190.

[9] Håvard Myrbakken and Ricardo Colomo-Palacios. DevSecOps: A Multivocal Literature Review. In Antonia Mas, Antoni Mesquida, Rory V. O'Connor, Terry Rout, and Alec Dorling, editors, *Software Process Improvement and Capability Determination*, Communications in Computer and Information Science, pages 17–29, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-67383-7_2.

[10] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE Software*, 33(3):94–100, May 2016. Conference Name: IEEE Software. doi:10.1109/MS.2016.68.

[11] Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. Continuous practices and devops: beyond the buzz, what does it all mean? pages 440–448, September 2017. doi:10.1109/SEAA.2017.8114695.

[12] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017. Conference Name: IEEE Access. doi:10.1109/ACCESS.2017.2685629.

[13] Daniel Teixeira, Ruben Pereira, Telmo Antonio Henriques, Miguel Silva, and João Faustino. A Systematic Literature Review on DevOps Capabilities and Areas. *International Journal of Human Capital and Information Technology Professionals (IJHCITP)*, 11(3):1–22, 2020. Publisher: IGI Global. URL: https://www.igi-global.com/gateway/article/www.igi-global.com/gateway/article/252844, doi:10.4018/IJHCITP.2020070101.

[14] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, January 2017. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215001430, doi:10.1016/j.jss.2015.06.063.

[15] CD Foundation. URL: https://cd.foundation/.

[16] GitHub features: the right tools for the job. URL: https://github.com/features.

[17] Iterate faster, innovate together. URL: https://about.gitlab.com/.

[18] Git - Documentation. URL: https://git-scm.com/doc.

[19] Jenkins User Documentation. URL: https://www.jenkins.io/doc/.

[20] Ansible Documentation — Ansible Documentation. URL: https://docs.ansible.com/ansible/latest/index.html.

[21] What is Docker?, June 2021. URL: https://www.ibm.com/cloud/learn/docker.

[22] Docker overview, February 2022. URL: https://docs.docker.com/get-started/overview/.

[23] What is Kubernetes? Section: docs. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[24] Kubernetes Documentation. URL: https://kubernetes.io/docs/home/.

[25] Sergii Telenyk, Oleksii Sopov, Eduard Zharikov, and Grzegorz Nowakowski. A Comparison of Kubernetes and Kubernetes-Compatible Platforms. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 313–317, September 2021. ISSN: 2770-4254. doi:10.1109/IDAACS53288.2021.9660392.

[26] Julien Vehent. *Securing DevOps: security in the Cloud*. Manning Publications Co, Shelter Island, New York, 2018. OCLC: on1050870710.

[27] OWASP Top 10:2021. URL: https://owasp.org/Top10/.

[28] Glossary. URL: https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/glossary.

[29] CWE - About - CWE Overview. URL: https://cwe.mitre.org/about/index.html.

[30] CVE - CVE-CWE-CAPEC Relationships. URL: https://cve.mitre.org/cve_cwe_capec_relationships.

[31] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, November 2004. Conference Name: IEEE Security Privacy. doi:10.1109/MSP.2004.111.

[32] Thorsten Rangnau, Remco v. Buijtenen, Frank Fransen, and Fatih Turkmen. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 145–154, October 2020. ISSN: 2325-6362. doi:10.1109/EDOC49727.2020.00026.

[33] Francesc Mateo Tudela, Juan-Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan-Antonio Sicilia Montalvo, and Michael I. Argyros. On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications. *Applied Sciences*, 10(24):9119, January 2020. Number: 24 Publisher: Multidisciplinary Digital Publishing Institute. URL: https://www.mdpi.com/2076-3417/10/24/9119, doi:10.3390/app10249119.

[34] Secure your application | GitLab. URL: https://docs.gitlab.com/ee/user/application_security/.

[35] What is eBPF? An Introduction and Deep Dive into the eBPF Technology. URL: https://ebpf.io/what-is-ebpf/.

[36] Runfeng Mao, He Zhang, Qiming Dai, Huang Huang, Guoping Rong, Haifeng Shen, Lianping Chen, and Kaixiang Lu. Preliminary Findings about DevSecOps from Grey Literature. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 450–457, December 2020. doi:10.1109/QRS51102.2020.00064.

[37] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Security as Culture: A Systematic Literature Review of DevSecOps. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 266–269. Association for Computing Machinery, New York, NY, USA, June 2020. URL: https://doi.org/10.1145/3387940.3392233.

[38] Isaac Dawson. How to benchmark security tools: a case study using WebGoat. URL: https://about.gitlab.com/blog/2020/08/11/how-to-benchmark-security-tools/.

[39] DORA. State of DevOps 2021. URL: https://services.google.com/fh/files/misc/state-of-devops-2021.pdf.