

DevSecOps: Integrating Security Testing into the Software Development Lifecycle

University of Bradford

Supervisor: Dr. Daniele Scrimieri

Student: Ahmed Olajide

UoB: 22012299

Dissertation Submitted In Partial Fulfillment of the Degree of
Msc in Cyber Security 2023

September 2023

ABSTRACT

In the ever-evolving landscape of software development, the integration of security practices has become paramount to safeguarding applications against a myriad of threats. This dissertation delves into the realm of DevSecOps—a philosophy that harmoniously blends development, security, and operations—offering a proactive approach to identifying and mitigating vulnerabilities throughout the software development lifecycle. The central objective of this research is to design, implement, and evaluate a DevSecOps pipeline that seamlessly integrates Static Application Security Testing (SAST), Software Composition Analysis (SCA), and Dynamic Application Security Testing (DAST) stages.

The dissertation presents a comprehensive exploration of DevSecOps principles, security testing techniques, and their integration into the software development lifecycle. The methodology encompasses the design of a DevSecOps pipeline architecture, tool selection, and the implementation details for each testing stage. Subsequently, the research offers a thorough analysis of the pipeline's performance, effectiveness, and its impact on early vulnerability detection and mitigation. The comparative analysis sheds light on the unique contributions of each security testing technique and underscores the importance of a balanced approach.

The findings reveal that the integrated DevSecOps pipeline empowers development, security, and operations teams to collaboratively address vulnerabilities before they escalate. The research underscores the significance of early detection and demonstrates the advantages of combining SAST, SCA, and DAST techniques for comprehensive vulnerability coverage. The study contributes to the DevSecOps discourse by offering a blueprint for organizations to implement effective security practices, fostering a culture of shared responsibility and secure software development.

Keywords: DevSecOps . DevOps . Security . Agile . Software Development Life-cycle . Security Analysis . Vulnerabilities

ACKNOWLEDGMENTS

Completing this dissertation would not have been possible without the guidance, support, and contributions of numerous individuals. I extend my heartfelt gratitude to my supervisor - Dr. Daniele Scrimieri, whose invaluable expertise and mentorship steered this research in the right direction. His encouragement, insightful feedback, and continuous guidance played an instrumental role in shaping this study.

I would also like to express my appreciation to Cyblack Community(A Non-Profit Organization that provides opportunities for Cyber security talents to kickstart and enhance their career), who provided resources to nurtured the growth of this research. The discussions, seminars, and interactions with fellow students greatly enriched my understanding of the subject matter.

Lastly, I want to acknowledge the unwavering support of my family(The Ayinlas'), friends, and wife, Olajide Oyindamola. Their encouragement, patience, and belief in my capabilities served as a constant source of motivation throughout this journey. Their understanding of the challenges and sacrifices associated with academic pursuits were pivotal in helping me overcome obstacles and reach this milestone.

This dissertation stands as a testament to the collaborative efforts and support of all those who have played a role, directly or indirectly, in its completion.

CONTENTS

Abstract	i
Acknowledgments	ii
Contents	v
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Research Aims and Objectives	3
1.2 Scope and Limitations	4
1.3 Significance of the Study and Project Outline	4
2 Literature Review	6
2.1 Theoretical Background	6
2.1.1 From DevOps To DevSecOps	8
2.1.2 DevSecOps and Regulatory Compliance	13
2.2 The Four Principle of DevSecOps	13
2.2.1 Culture	14
2.2.2 Automation	15
2.2.3 Measurement	15
2.2.4 Sharing	15
2.3 Key Security Testing Techniques for DevSecOps	16
2.3.1 Static Application Security Testing (SAST)	16
2.3.2 Dynamic Application Security Testing (DAST)	17
2.3.3 Interactive Application Security Testing (IAST)	17
2.3.4 Software Composition Analysis (SCA)	18
2.3.5 Container Security Scanning	18
2.4 Cloud Infrastructure and Security	19
2.4.1 Advantages of Cloud Infrastructure for DevSecOps Practices	19
2.4.2 Security Considerations in Cloud Environments	20
2.5 Tools Selection and Rationale	22
2.5.1 SAST Tooling	22

2.5.2	DAST Tooling	22
2.5.3	SCA Tooling	24
2.5.4	CI/CD Tooling	24
2.5.5	Infrastructure as Code (IaC) Tooling	25
2.5.6	Container Platforms	26
2.5.7	Cloud Environment	27
2.5.8	Infrastructure Setup	30
2.6	Research Gaps and Challenges	30
2.7	Conclusion	31
3	Methodology	32
3.1	Research Design and Approach	32
3.2	End to End Case Study That Will Be Implemented	32
3.3	Cloud Infrastructure	34
3.4	CI/CD Software	39
3.4.1	Jenkins and Its Plugins	39
3.4.2	Project Compilation With Maven	40
3.4.3	Git Repository	41
3.4.4	Container Image Compilation With Docker	43
3.4.5	Kubernetes Orchestration Integration	43
3.5	Conclusion	45
4	Implementation and Results	46
4.1	Target Web Application	46
4.2	SAST Implementation	46
4.2.1	Parameters for SonarCloud Configuration	47
4.2.2	SonarCloud Build and Test Result	47
4.3	SCA Implementation	49
4.3.1	Parameters for Synk Configuration	49
4.3.2	Snyk Build and Test result	50
4.4	Building and Containerizing Application	51
4.5	Seamless Deployment of Containerized Application on AWS Kubernetes	52
4.6	DAST Implementation	55
4.6.1	Parameters for OWASP ZAP	55
4.6.2	OWASP ZAP Build and Test Result	56
4.7	Discussion of Result	57
4.7.1	Dicovered Vulnerabiities	57
4.7.2	Comparison of The Security Testing Techniques	58
4.8	Conclusion	60

5	Conclusion and Future Work	62
5.1	Conclusion	62
5.2	Limitations and Future Work	63
5.2.1	Limitations of the Study	63
5.2.2	Future Work	63
	References	65
A	An Appendix	72
A.1	Deployment.yaml	72
A.2	Jenkinsfile	73
A.3	Dockerfile	75
A.4	OWASP ZAP Result	75
A.5	Terraform Declarative File	76
A.6	Shell Script	78

LIST OF FIGURES

1.1	Building security into a DevOps pipeline (Das & Chu 2023)	3
2.1	Software Development Life Cycle (SDLC). Waterfall Model (Sarycheva, 2019)	7
2.2	DevOps Lifecycle (Mohanam, 2022)	10
2.3	DevSecOps Lifecycle (Gartner, 2016)	12
2.4	Image showing some services offered by AWS (Logz n.d.)	29
3.1	Comprehensive Illustration of the DevSecOps Pipeline to be Deployed	34
3.2	Configuring the AWS CLI for the IAM user for programmatic access	35
3.3	The Terraform main.tf code snippet written in HashiCorp Configuration Language (HCL)	36
3.4	Image showing the ports opened to allow traffic as port 8081 is for jenkins and 22 for SSH	36
3.5	Using terraform init to initialize a terraform Working Directory	37
3.6	Image showing the resources that will be created within AWS using terraform plan	37
3.7	Creating the aws resources as declared in the terraform main.tf configuration file using terraform apply	38
3.8	Running virtual machine created on AWS with Jenkins, Kubernetes, Docker OWASP ZAP and Maven installed	38
3.9	Jenkins successfully installed and accessed through port 8081	39
3.10	Installation of important plugins on Jenkins	40
3.11	Image showing maven version 3.5.2 installed on the virtual machine	41
3.12	Maven plugin being configured on Jenkins with the maven home directory	41
3.13	Image showing the hosted java application on GitLab repository where jenkins pulls from	42
3.14	Image of the dockerfile using the official Maven 3.8 image with JDK 8 to build our application	43
3.15	Creating a Kubernetes Cluster on AWS	44
3.16	Deployment.yaml Code Snippet	44
4.1	SonarCloud Pipeline Stage Implementation In Jenkinsfile	47
4.2	SonarCloud Pipeline Build Result In Jenkins	48
4.3	SonarCloud Vulnerability Scan Result	48

4.4	Image Showing SonarCloud Recommended Solution To Fix the Code Issue	49
4.5	Synk Pipeline Stage Implementation In Jenkinsfile	49
4.6	Synk Pipeline Build Result In Jenkins Dashboard	50
4.7	Snyk Scan Result In Jenkins Console Output	50
4.8	Code Snippet of The Build and Push Stage in the Jenkinsfile	51
4.9	Image Showing A successful Jenkins Pipeline With The App Image Built and Push to ECR	52
4.10	Image of The App Dockerized Image Built to Amazon ECR Ready For De- ployment to Amazon Kubernetes	52
4.11	Image of The App Dockerized Image Built to Amazon ECR Ready For De- ployment to Amazon Kubernetes	53
4.12	Successful Deployment of EasyBuggy Application on Kubernetes	53
4.13	Successful Deployment of EasyBuggy Application on Kubernetes	53
4.14	EasyBuggy Web Application Up and Running on Kubernetes Cluster	54
4.15	EasyBuggy Web Application Up and Running on Kubernetes Cluster	55
4.16	OWASP ZAP Pipeline Build Result on Jenkins	56
4.17	OWASP ZAP Scan Test Result	56
A.1	Configuration file for eassybuggy containerized applications on Kubernetes	72
A.2	Configuration file for eassybuggy containerized applications on Kubernetes - cont'd	73
A.3	Jenkins configuration file for security build stages	73
A.4	Jenkins configuration file for security build stages-cont'd	74
A.5	Jenkins configuration file for security build stages-cont'd	74
A.6	Config file used to build the docker container image for the application	75
A.7	Vulnerabilty result and its associated CWE ID	75
A.8	Vulnerabilty result and its associated CWE ID - cont'd	75
A.9	Terraform script used for deploying AWS resources	76
A.10	Terraform script used for deploying AWS resources - cont'd	76
A.11	Terraform script used for deploying AWS resources - cont'd	77
A.12	Terraform script used for deploying AWS resources - cont'd	77
A.13	Terraform script used for deploying AWS resources - cont'd	78
A.14	Shell script used for installing softwares on AWS EC2	78
A.15	Shell script used for installing softwares on AWS EC2 - cont'd	79

LIST OF TABLES

4.1	Advantages and Disadvantages of Each Security Testing Techniques . . .	60
-----	--	----

Chapter 1

INTRODUCTION

Software plays a vital role in the functioning of organizations across different sectors in the current digital environment. Nevertheless, the continuous advancement of technology and the increasing complexity of cybersecurity risks have revealed weaknesses in software systems, resulting in security breaches and jeopardizing confidential information. In recent years, there has been a shift in focus among software development companies from developing software as a product (SaaP) to developing software as a service (SaaS). Previously, companies would develop the software and deliver a finished product to customers who would then install and run it locally. However, with SaaS, the software is centrally hosted on a cloud infrastructure and accessed directly by customers through channels such as web browsers (Mell & Grance, 2011) or other means that delivers it directly to customer's device's (Fitzgerald & Stol, 2017).

The use of SaaS is made available to customers through licensing and subscriptions. In this model, customers do not have control over the underlying cloud infrastructure or the functionality of the application (Mell & Grance, 2011). These aspects are managed by the provider, allowing them to continuously enhance and deliver their software without needing to redistribute it to all clients. Instead, they can simply update the software on their own cloud infrastructure. This feature enables the software provider to seamlessly enhance and deploy their software without the requirement of redistributing it to each client. By updating the software on their own cloud infrastructure, they can continuously enhance and distribute it. This modern software engineering approach entails the complex process of developing, integrating, and distributing software in a continuous manner called Continuous integration (CI) and Continuous Delivery (CD).

Continuous integration (CI) is the process of automatically integrating new code from multiple developers into the current version of the software while simultaneously conducting error check (Albaihaqi et al., 2020). **Continuous Delivery (CD)** is a practice in which

software changes are regularly and reliably released to the production or live environment. It entails automating the software's build, testing, and deployment process to facilitate quick and efficient delivery to users. By adopting continuous delivery, teams can maintain their software in a constantly releasable state, enabling them to regularly and consistently provide new features, bug fixes, and enhancements to users without the need for manual interventions or delays (Bolscher & Daneva, 2019).

A significant assortment of tools and information systems is necessary for these processes (Ebert et al. 2016). Independent operations teams typically handle the management of these processes, tools, and systems (Cois et al., 2014). Lack of collaboration and communication between operators and developers has led to numerous challenges in implementing CI/CD. In an effort to address these challenges, the concept of DevOps has emerged (Fitzgerald & Stol, 2017).

This concept is grounded in the collaborative efforts of two former fields throughout all stages of development. This collaboration is achieved through problem-solving together, automating processes, and agreeing upon shared metrics for evaluating a system. DevOps encompasses four pillars that guide teamwork in modern software development: **culture, automation, measurement, and sharing** (CAMS) (Humble & Molesky, 2011). By adopting this agile development method, software practitioners are able to test and deploy software versions at a much faster rate, allowing for rapid response to customer demands. A notable example of this is demonstrated by Amazon, where new versions were released at a frequency of more than once per second (Yasar & Kontostathis, 2016).

While, fast releases are generally seen as advantageous for product quality, but they can also create additional pressure for developers to complete their tasks at a faster pace. Research conducted by Kraemer et al. (2009) has demonstrated that tight schedules and heavy workloads can inadvertently result in the introduction of security vulnerabilities in software systems. Kraemer further asserts that the lack of security knowledge among DevOps teams is a contributing factor to the existence of these vulnerabilities, which consequently impacts the efficacy of security tests and compromises the overall security of a system.

The integration of security measures in the DevOps process has posed a difficulty due to the inability of traditional security approaches to match the agility and speed of DevOps. DevSecOps expands the objective of incorporating security into DevOps, which already emphasizes quality and speed. The primary focus in integrating security into DevOps has been on introducing more tools and automating test-driven security within the CI/CD

pipeline. DevSecOps has facilitated the collaboration of development, security, and operations teams in order to incorporate security practices into every stage of the software development life-cycle (Myrbakken and Colomo-Palacios 2017). An important component of this integration is the practice of Static Application Security Testing (SAST), Software Composition Analysis (SCA), and Dynamic Application Security Testing (DAST), which entails early identification of potential vulnerabilities and risks in the development process for efficient mitigation (Akbar et al. 2022).

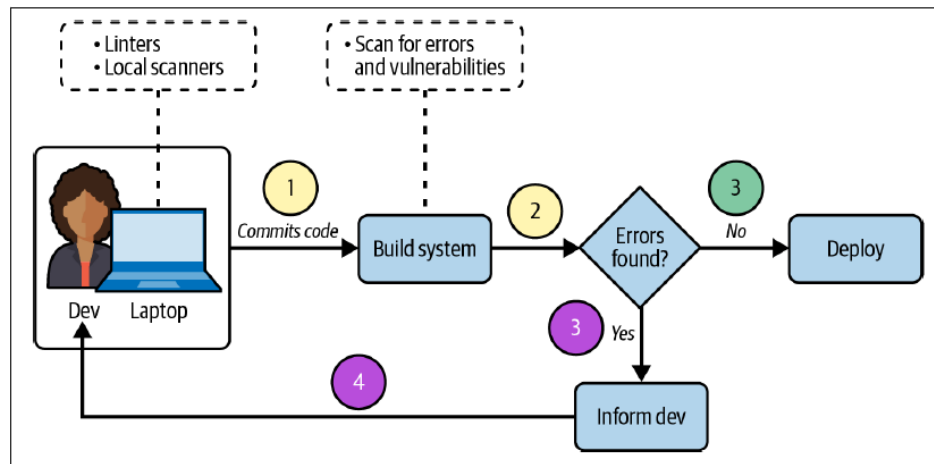


Figure 1.1: Building security into a DevOps pipeline (Das & Chu 2023)

1.1 Research Aims and Objectives

The primary objective of this research is to design and implement a comprehensive DevSecOps pipeline that seamlessly integrates security practices into the software development lifecycle. The objectives and aims are:

1. To create a blueprint for constructing a DevSecOps pipeline that brings together development, security, and operations seamlessly. This blueprint will consider the specific needs of the software development process and outline how different security tests can be integrated at various stages, from code writing to deployment.
2. To construct and implement a DevSecOps pipeline that seamlessly integrates Static Application Security Testing (SAST), Software Composition Analysis (SCA), and Dynamic Application Security Testing (DAST) stages. This seeks to demonstrate how

the collaborative efforts of development, security, and operations teams can proactively detect and mitigate vulnerabilities.

3. To assess the efficacy of the selected security testing tools, including SonarCloud, Snyk, and OWASP Zap, in identifying and addressing vulnerabilities. This objective aims to quantify the impact of each testing technique and identify their strengths and limitations in DevSecOps.
4. To orchestrate a robust and scalable cloud infrastructure in the AWS environment through Terraform and Kubernetes, facilitating the seamless deployment and management of applications within the DevSecOps pipeline.
5. To identify research gaps in the domain of DevSecOps and software security. This objective involves comparing the findings of this research with existing literature, pinpointing areas where further exploration is needed, and proposing future research directions.

1.2 Scope and Limitations

This research is primarily focused on the integration of security practices into the software development lifecycle through the implementation of DevSecOps principles. The study encompasses the integration of SAST, SCA, and DAST within a Jenkins-based CI/CD pipeline. The AWS cloud environment is chosen for the deployment of the required infrastructure using Terraform and Kubernetes. However, it's important to acknowledge that this study does not delve into every possible security testing technique or cloud environment.

The limitations of this study include the scope of tools and methodologies employed. While SAST, SCA, and DAST are effective security testing techniques, there are other practices and tools that contribute to a holistic security approach. Moreover, the focus on AWS as the cloud environment excludes considerations specific to other cloud providers.

1.3 Significance of the Study and Project Outline

The significance of this research is threefold. Firstly, it addresses a critical gap in the current software development landscape by proposing a comprehensive approach to se-

curity integration. Secondly, by employing real-world tools like Jenkins, Terraform, and Kubernetes, this study provides practical insights into the implementation process, aiding practitioners in replicating the approach. Lastly, the study's outcomes are expected to contribute to the broader understanding of how security practices impact software development efficiency, application robustness, and the overall risk landscape.

This dissertation follows a structured format. In Chapter 1, an introduction is provided on the research topic, including background information, the four pillars of DevSecOps. Additionally, an overview is given on the current state of security in software development, along with the research objectives and the scope of this research. Chapter 2 comprises a comprehensive literature review on DevSecOps, security testing methodologies, and existing research and best practices in integrating cloud environments within DevSecOps. Moving on to Chapter 3, the methodology employed in this research is described, which includes explaining the research approach, and the tools that will be used for automation and scanning for vulnerabilities within the pipeline. The chosen methodology is justified based on its suitability for addressing the research objectives. In Chapter 4, a detailed account is given of how the security testing approach was implemented within a DevSecOps environment. It presents the results and findings from these experiments, evaluates the effectiveness and efficiency of this approach. Chapter 5 serves as a discussion and conclusion section for the research findings. It analyzes the implications and contributions of this study, identifies any limitations encountered during the research.

In conclusion, this chapter has laid the foundation for the research by outlining the problem of inadequate security in the software development lifecycle, articulating research objectives and questions, defining the scope and limitations of the study, and emphasizing its significance. The subsequent chapters will delve deeper into the literature surrounding DevSecOps, security testing methodologies, cloud infrastructure, and the implementation process. Through this exploration, the research aims to provide valuable insights into securing software applications in the face of escalating cyber threats.

Chapter 2

LITERATURE REVIEW

DevSecOps addresses the requirement for security in the field of DevOps. It aims to integrate contemporary security practices into the fast-paced and flexible environment of DevOps. By involving security experts from the beginning, it extends DevOps' objective of fostering collaboration between developers and operators (Mohan & Othmane 2016).

Given that DevSecOps is a recent trend, it is crucial to gain an understanding of the practices and knowledge that have been accumulated in this area. Although there is limited research available on DevSecOps, a review of the existing literature reveals the following: one study examines practitioners' experiences and the practices they engage in through Internet artifacts and a survey (Ur Rahman & Williams 2016), while another study provides a systematic mapping (SM) of the research being conducted in this field. The systematic mapping study highlights that research is currently focused on various aspects such as defining DevSecOps, identifying security best practices, ensuring compliance, automating processes, developing tools for DevSecOps, managing software configuration, facilitating team collaboration, capturing activity data, and maintaining information secrecy (Mohan & Othmane 2016).

2.1 Theoretical Background

The process of software reaching our laptops or mobile devices is thorough and intricate. It requires several stages of design and development before a usable product is produced, following the acquisition of a new software idea. Due to the complexity and abstract nature of modern software, which typically have multiple functions, it is essential for them to be delivered to customers with quality and security assured. Consequently, the journey from concept to product can be lengthy. To ensure a systematic and well-structured approach

to software development, a development method is employed. This process is commonly referred to as the Software Development Life Cycle (SDLC).

In the initial stages of one's professional career, there is a sense of satisfaction in successfully creating code that fulfills the specified requirements. However, as time progresses, it becomes evident that writing high-quality code involves more than simply solving basic arithmetic problems. It requires considering various scenarios, such as handling user input errors. To illustrate this point, let's consider the scenario of a calculator program encountering the input "2 + a". If as a programmer, you overlooked this particular use case because it was not explicitly mentioned in the requirements and it was not included in the testing process conducted by your quality assurance team, you would end up delivering faulty code to your customers. This flawed code would be deemed as the final product, which would ultimately hinder the user experience rather than enhance it (Das & Chu, 2023).

The Software Development Life Cycle (SDLC) model is a widely used representation of the flow described in the text (see Figure 1.1). In practical terms, the SDLC's waterfall model operates as follows: developers swiftly create code based on functional requirements, which is then sent for testing. During testing, errors are identified and the code is returned to the developers for fixes. The corrected code undergoes another round of testing. Upon completion of testing, the code is transferred to the operations team for maintenance (Sarycheva, 2019).

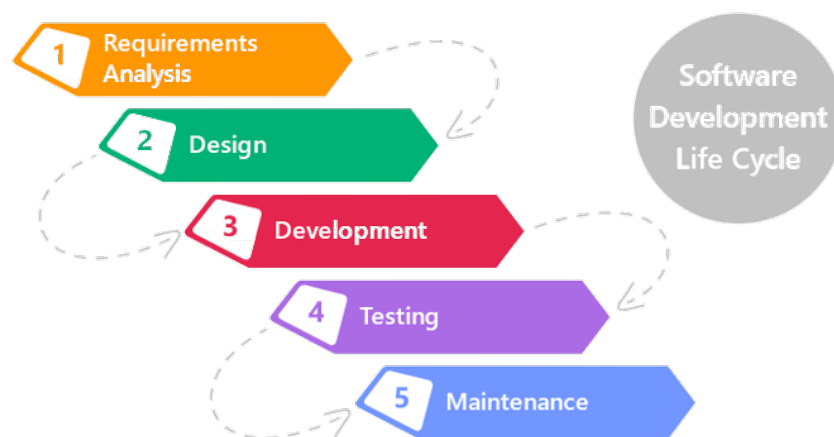


Figure 2.1: Software Development Life Cycle (SDLC). Waterfall Model (Sarycheva, 2019)

The waterfall model fosters extensive communication and coordination among teams, as previously mentioned. This exchange becomes more challenging when no deliverables

are produced due to code failing testing, resulting in a significant waste of time and effort without yielding any concrete results. This is where DevOps come into play.

In 2012, Neil MacDonald coined the term **DevOpsSec** with the aim of incorporating security into DevOps practices without compromising speed and agility. DevSecOps, as a methodology, involves integrating security throughout the entire application development life cycle rather than treating it as an afterthought (Kumar & Goyal 2020). Many organizations are embracing DevOps and DevSecOps to enhance their Software Development Life Cycle (SDLC) and improve the overall efficiency, collaboration, and security of their software development processes.

Carturan & Goya (2019) emphasize the critical importance of incorporating security verification and validation throughout the software development process. Their study highlights that secure software development involves various factors, with security being a central concern. In their research, the authors introduce a comprehensive security framework designed to guide organizations during the planning and definition stages of security requirements. This framework takes into consideration the agile methodologies commonly employed in modern application development.

2.1.1 From DevOps To DevSecOps

DevOps and DevSecOps are two distinct methodologies used in software development and deployment, each with its own specific areas of emphasis and objectives. To begin with, we will provide a concise overview of DevOps before delving into the concept of DevSecOps.

Understanding DevOps

DevOps is a methodology that integrates development(Dev) and operations(Ops), bringing together individuals, processes, and technology in various stages of application management. According to Leite et al. (2019), DevOps is an interdisciplinary and collaborative approach in organizations that aims to automate the continuous delivery of software updates, ensuring their accuracy and dependability. The development team, along with stakeholders from customer service, operations, and quality assurance, work together to consistently release software products, capitalize on market opportunities, and decrease the time required to incorporate customer feedback. Ultimately, DevOps facilitates the continuous delivery of valuable and dependable products to customers.

A DevOps process would have the following phases: (Candel, 2022)

- A software developer utilizes their preferred development environment to write the code. Subsequently, they proceed to upload the code to a centralized code repository such as Git or Bitbucket.
- The Continuous Integration (CI) server retrieves the source code from the central repository and compiles it into artifacts and binaries. This process includes creating Docker images and uploading them to the Docker registry specifically for containerized applications.
- The artifacts and binaries obtained from the repository are transferred to various pre-production and production environments for deployment. In these environments, container technologies such as Docker and Kubernetes are utilized to build them.
- Docker images are used to start containers. In non-containerized environments, such as Virtual Machines (VMs), the process may involve simply copying the binaries to a specified location.

DevOps has the potential to enhance application security through the implementation of various best practices. Some of these practices include: (Adnan et al., 2023)

- Incorporate automated security testing methods, such as fuzz testing and software penetration testing, into the software development lifecycle or system integration cycle.
- Standardization of the integration cycle to reduce the introduction of errors.
- At the commencement of projects, software and systems development teams are informed about security issues and limitations.

The DevOps lifecycle enhances development processes from initiation to completion and involves the organization in ongoing development, leading to accelerated delivery times. This approach primarily encompasses seven distinct stages (Mohanani, 2022) as shown Figure 1.2 below

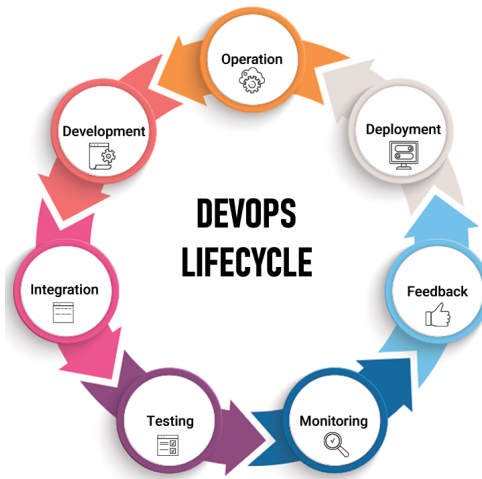


Figure 2.2: DevOps Lifecycle (Mohanana, 2022)

Now, let's move on to DevSecOps, which stands for Development, Security, and Operations. DevSecOps extends the principles of DevOps by integrating security practices into the software development process from the beginning. It aims to make security an integral part of the development pipeline, rather than a separate step at the end (Microsoft, 2022).

Now To DevSecOps

The efficient and timely release of software products by organizations, particularly with the implementation of DevOps, necessitates the utilization of appropriate tools and processes. In such scenarios, DevSecOps proves to be highly advantageous to organizations as it integrates privacy and security measures into DevOps practices, while enabling operations to continue with a heightened level of cybersecurity (Candel, 2022).

The primary objective of DevSecOps is to guarantee that security is taken into account and dealt with at every stage of the software development process, thus facilitating the creation of software that is both secure and resilient. This approach fosters a culture of collective responsibility among developers, security experts, and operations teams, ensuring that all individuals are held accountable for ensuring security (Myrbakken & Colomo-Palacios, 2017).

DevSecOps can be defined as a framework that incorporates security activities into the software development and operational process, including requirements gathering, design, coding, testing, delivery, deployment, and incident response (Yasar, 2020). Developers are encouraged to consider security aspects during the design and writing of code, rather

than considering it as an afterthought. This change in mindset aids in the early detection and resolution of security issues, thereby decreasing the chances of vulnerabilities being present in the final product. This approach is commonly referred to as the "**Shift-Left**" security mindset (Carter 2017).

Rajapakse et al. (2022) offer valuable insights into the challenges faced by practitioners during the adoption of DevSecOps. They emphasize the significance of addressing these challenges to promote a successful DevSecOps transition. These challenges primarily revolve around the integration of security practices into the rapid-paced DevOps Software Development Life Cycle (SDLC). The authors highlight several key challenges and propose solutions derived from existing research which one is the "**Shift-Left**" security mindset.

To implement a shift towards left in terms of security, it is essential to prioritize early practices in the software development life cycle. This shift can be achieved through the utilization of automated tools like static analysis. Integrating security tools into the DevSecOps pipeline facilitates increased developer self-service by allowing them to efficiently check the security of their product. Shifting left also offers additional advantages, such as enhancing the security and overall quality of the product. Developers are consistently faced with the challenge of producing larger quantities of software in shorter timeframes. It is crucial to avoid the occurrence of code failures towards the end of the development life cycle. The efficiency of addressing and rectifying flaws in the software development life cycle (SDLC) significantly improves when developers are able to promptly detect, assess, and correct such issues (Curphey 2019).

Khan (2020) provides an overview of the secure DevOps workflow and explains how organizations can incorporate continuous security testing into their Continuous delivery pipeline. This work is highly significant in terms of integrating security measures within a DevOps environment. Khan examines various security controls, tools, automated checks/testing, and best practices to ensure thorough software testing throughout the development.

Mature DevOps practices involve the continuous testing, deployment, and validation of software to ensure that it fulfills all requirements and enables prompt recovery in case of issues. Consequently, it can be asserted that DevSecOps represents an optimized implementation of DevOps principles (Yasar, 2020).

These are the practices of how DevSecOps is implemented:(Candel, 2022)

- The integration of security tools into the development integration process.
- Give priority to security requirements when considering the product's backlog.
- Collaborate with the security and development teams on the threat model
- Review infrastructure-related security policies prior to deployment

In such situations, organizations should consider implementing a DevSecOps methodology as it effectively incorporates industry best practices throughout the software product development cycle. This approach involves integrating security measures into all aspects of software development, encompassing infrastructure, continuous integration, deployments, and continuous delivery of applications. Furthermore, it is essential for applications to adhere to information security best practices, encompassing aspects such as data integrity, availability, and confidentiality. By doing so, developers gain knowledge on secure coding techniques and the importance of understanding security best practices (Candel, 2022).

DevOps recommends incorporating various tools to verify the accuracy of code during the development cycle. Similarly, DevSecOps proposes integrating security tools into the continuous integration and deployment processes (Parashar et al., 2020). These tools are also known as application security pipelines (Parashar et al., 2020). These pipelines encompass stages such as automated code review, security testing, scans, monitoring, and automated report generation. This represents the life cycle and sequence of phases within the DevSecOps ecosystem where security measures are integrated throughout the entire process (Guo, 2021).

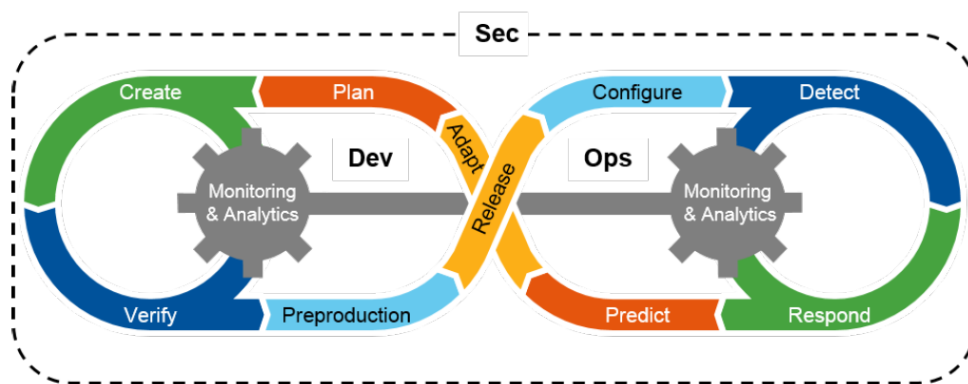


Figure 2.3: DevSecOps Lifecycle (Gartner, 2016)

DevSecOps implementation allows for the automation of security functions like Identity

and Access Management (IAM), firewall management, and vulnerability scanning across the DevOps lifecycle. As a result, security teams can focus on policy creation and enforcement without being burdened by manual security tasks (Gartner, 2016). Integrating security into the development process, rather than adding it as a separate layer afterwards, enables DevOps and security professionals to leverage agile methodologies. This integration aims to prevent obstacles and facilitate the production of secure code (Gartner, 2016).

2.1.2 DevSecOps and Regulatory Compliance

Jensen et al. (2019) and Hsu (2018) contribute insights into the alignment of DevSecOps practices with regulatory compliance requirements and the broader enhancement of organizational security:

Efficient Compliance with Legal Frameworks: Jensen et al., (2019) work explores how DevSecOps practices can efficiently ensure compliance with regulatory frameworks such as GDPR and HIPAA. Their research sheds light on the pivotal role of DevSecOps in aligning software development with legal and regulatory standards. This alignment is instrumental in safeguarding sensitive data and ensuring that organizations meet their legal obligations.

Enhancing Organizational Security through DevOps Integration: Hsu (2018) underscores the significance of integrating DevOps strategies not only for infrastructure safeguarding but also for enhancing security at all levels within an organization. Hsu's case studies illustrate the real-world utilization of DevSecOps practices to not only adhere to GDPR but also to securely handle Personally Identifiable Information (PII). This integration of DevOps principles with security measures continually bolsters an organization's security posture.

2.2 The Four Principle of DevSecOps

The four principles of the DevOps, namely culture, automation, measurement, and sharing (abbreviated as CAMS), were initially presented by DevOps pioneer John Willis (2010) in a blog post that explored the fundamental values of DevOps development. Since their

introduction, these CAMS principles have been widely adopted as a theoretical framework in various academic studies (e.g., Myrbakken & Colomo-Pacios 2017, Humble & Molesky 2011) and industry publications (e.g., Puppet 2019).

These principles were later adapted and extended to DevSecOps, where the "Security" aspect was integrated to emphasize the importance of incorporating security practices into the DevOps culture (Rangnau et al. 2020).

The research conducted by Diel et al., (2016) highlights the importance of collective team-work in the development of secure products, rather than solely relying on the security team. This concept, known as "DevSecOps," suggests incorporating security practices into an organization's culture, with every team committing to adhering to security procedures and sharing equal responsibility for product security.

Here's a breakdown of how the four principles relate to both DevOps and DevSecOps

2.2.1 Culture

The DevSecOps culture aims to foster collaboration and shared responsibility among the development, security, and operations teams. It emphasizes more than just the implementation of security tools and technologies, but also focuses on creating a development process where security is deeply embedded in every aspect. This cultural principle addresses the mindset, attitudes, and behaviors of individuals involved in the software development lifecycle, promoting a collective accountability for security (Rangnau et al. 2020). DevSecOps eliminate the traditional practice of separate departments. Unlike the traditional software development model, where development is completed by one team and then passed on to operations and security teams, DevSecOps promotes shared responsibility (Carter, 2017). This means that any defects or issues are seen as collective problems. In contrast, the traditional model involves development teams solely focusing on software development and then handing it off to operations for testing and security team for evaluation and vulnerabilities.

2.2.2 Automation

The principle of "Automation" in DevSecOps is a fundamental belief that emphasizes the strategic utilization of technology to streamline and enhance the integration of security practices throughout the software development lifecycle. Automation plays a critical role in incorporating security into the development process by reducing manual effort, ensuring consistency, and facilitating a more efficient and agile delivery of secure software (Garousi & Mäntylä 2016). However, it is important to note that while test automation is valuable, it cannot fully replace manual testing. This is due to the fact that it is not possible to automate all test cases. In certain scenarios, manual checks are indispensable as they can detect errors or issues that automated testing tools are unable to identify, such as authentication and authorization problems (Kumar & Goyal 2020).

2.2.3 Measurement

DevSecOps places significant emphasis on the collection and analysis of relevant metrics and data to assess the effectiveness of security practices throughout the software development lifecycle. Its core principle is that improvement is impossible without measurement (Yasar, 2018). By consistently monitoring and evaluating key performance indicators (KPIs) related to security, organizations can gain valuable insights into their security posture, identify areas for enhancement, and make informed decisions to improve their DevSecOps processes (Garousi & Mäntylä, 2016). The measurement process should encompass all stages of development and incorporate metrics from operational monitoring. Additionally, it is crucial for the organization to not only measure but also utilize the resulting data effectively. For example, feedback from security monitoring should be provided to developers in order to facilitate product improvement.

2.2.4 Sharing

Sharing is a crucial principle of DevSecOps, as it ensures stability in the practice. The adoption and performance of DevSecOps rely on the sharing of both successes and failures (Puppet, 2019). When failures are not shared, teams are left to face the same obstacles independently. Similarly, when successes are not shared, valuable practices remain confined to the teams that originated them. To facilitate growth and benefit the entire or-

ganization, sharing must be emphasized throughout. By constantly sharing challenges and offering assistance, teams can improve security processes (Carter, 2017). Collaboration among developers, operations, and security personnel is essential for creating a high-quality product. Taking a sharing mindset avoids dealing with problems in isolated silos and instead promotes collective problem-solving efforts. Feedback from the product's behavior in production informs developers, while input from developers aids in addressing security issues effectively. For instance, if a security concern arises during development, developers can collaborate with information security team members to determine the most suitable solution, such as implementing a firewall or making design changes (Mansfield-Devine 2018).

2.3 Key Security Testing Techniques for DevSecOps

In the previous section, we gained an understanding of DevSecOps and the significance of integrating security testing into the software development lifecycle. Now, let's explore the key security testing techniques that play a crucial role in DevSecOps. These techniques help identify and address security vulnerabilities at different stages of the development process, ensuring that applications are robust, resilient, and less susceptible to cyber threats. The integration of security testing tools into CI/CD pipelines is a critical aspect of DevSecOps (Rangnau et al., 2020). This enables the rapid detection and mitigation of vulnerabilities throughout the development process, aligning with the principles of continuous integration and continuous delivery.

2.3.1 Static Application Security Testing (SAST)

SAST, which is also known as **white-box** test is a technique that involves analyzing the source code or application's binary without executing it. The primary objective of SAST is to identify potential security flaws in the codebase early in the development lifecycle. By analyzing the code statically, SAST tools can detect vulnerabilities such as code injection, cross-site scripting (XSS), SQL injection, and insecure data handling (Dhawan & Sabetto 2019). Integrating SAST into the DevSecOps pipeline allows developers to receive immediate feedback on security issues in their code. Early detection of vulnerabilities empowers developers to remediate them quickly, reducing the cost and effort of fixing security flaws later in the development process.

SAST tools provide feedback to developers on identified security issues. The feedback typically includes detailed information about the vulnerabilities found, including location in the code and recommended fixes. Developers can then prioritize and address the issues accordingly (MacDonald & Head 2016).

2.3.2 Dynamic Application Security Testing (DAST)

DAST, complements SAST by focusing on testing the application while it is running. Instead of analyzing the code itself, DAST interacts with the application to identify security weaknesses from a **black-box** perspective. It helps to evaluate the security posture of an application by simulating real-world attacks, attempting to exploit potential vulnerabilities like input validation flaws, insecure configurations, and authentication issues (Candel, 2022).

The involvement of the security team is essential in the process of developing web applications. They are responsible for conducting on-demand and manual security evaluations using DAST tools to identify any vulnerabilities in each new version or update. This step is crucial in order to ensure that the deployment of applications aligns with the organization's secure software development methodology (Candel, 2022).

2.3.3 Interactive Application Security Testing (IAST)

IAST is an advanced security testing technique that combines elements of both SAST and DAST. IAST instruments the application to monitor its behavior during runtime. By observing the application's execution, IAST identifies vulnerabilities in real-time with contextual information. The unique advantage of IAST is its ability to provide deeper insights into security issues (Candel, 2022). It offers developers valuable context, including data flow paths and variable values that led to the security vulnerability. IAST's ability to pinpoint the root cause of vulnerabilities helps developers understand and address security issues more efficiently.

IAST may not cover all types of security vulnerabilities, and it is essential to use it in conjunction with other security testing techniques like SAST and DAST for comprehensive coverage. IAST requires proper configuration and tuning to minimize false positives and negatives (Tudela et al., 2020).

2.3.4 Software Composition Analysis (SCA)

SCA, is a security testing technique that focuses on identifying vulnerabilities in third-party and open-source components used in the application. Many modern applications heavily rely on external libraries and frameworks. However, these dependencies can introduce vulnerabilities, which may go unnoticed without proper scrutiny (Snyk, 2021).

Imagine a scenario where a company develops an application for its customers. They use various open-source libraries to expedite development and enhance functionality. However, these libraries may have vulnerabilities or licensing restrictions that could pose risks to the company and its users. This is where SCA comes into play. It enables organizations to identify all the components used in their software, including open-source libraries, frameworks, and even code snippets copied from other sources. By conducting a thorough analysis, SCA tools can provide insights into any known vulnerabilities or licensing issues associated with these components (Snyk, 2021).

Integrating SCA into the DevSecOps pipeline ensures that developers are aware of any risky components they are using, enabling them to make informed decisions about their dependencies and apply necessary updates or patches.

2.3.5 Container Security Scanning

Containers have revolutionized application deployment and scalability, but they also introduce unique security challenges. Container security scanning involves analyzing container images for known vulnerabilities and misconfigurations before they are deployed (Souppaya, 2017).

DevSecOps teams can integrate container security scanning into their CI/CD pipelines to automatically check container images for security flaws. This proactive approach ensures that only secure and trusted container images are deployed, reducing the risk of exposing the application to potential threats. With the increasing popularity of containerization technologies like Docker and Kubernetes, proper container security scanning has become even more critical. It provides peace of mind to businesses and users alike, knowing that their applications are shielded from potential threats (Souppaya, 2017).

Van Wyk & McGraw (2005) emphasize the importance of integrating security techniques

into CI/CD practices to ensure high security quality in software systems. Their work focuses on automating security testing, including both static and dynamic methods, within the CI/CD pipeline.

2.4 Cloud Infrastructure and Security

Cloud computing is becoming more popular among organizations for its many advantages. One important aspect to consider is the relationship between cloud infrastructure and security in the context of DevSecOps. Companies, such as Amazon Web Services (AWS), offer scalability, flexibility, and cost-effectiveness through their cloud environments. However, it is essential to acknowledge that these environments also bring about specific security concerns that need to be addressed (Omer 2014).

2.4.1 Advantages of Cloud Infrastructure for DevSecOps Practices

One of the key enablers of successful DevSecOps implementation is cloud infrastructure, with Amazon Web Services (AWS) being a prominent player in this arena (Mishra 2023). In this section, we will delve into the myriad benefits that cloud infrastructure, like AWS, offers for effective DevSecOps practices.

Scalability is a fundamental aspect of cloud infrastructure, which is crucial for meeting the demands of DevSecOps. Cloud platforms, such as AWS, are highly proficient in efficiently managing diverse workloads. With AWS, teams can effortlessly adjust their infrastructure to accommodate fluctuating demand, thereby ensuring optimal performance and resource utilization consistently (Mishra 2023). DevSecOps is not a one-size-fits-all practice. Different organizations have distinct needs and preferences when it comes to tools and services. Cloud platforms, offer a wide array of customizable tools that can be seamlessly integrated into DevSecOps pipelines. This **flexibility** empowers teams to select and tailor tools that align perfectly with their unique requirements (Saini et al., 2019).

Automation plays a crucial role in the implementation of DevSecOps. Cloud platforms offer strong automation functionalities that enhance efficiency, minimize human mistakes, and enforce consistent deployment and security procedures. By leveraging AWS's automation features, teams are able to improve effectiveness and dependability throughout

the entire DevSecOps cycle (Saini et al. 2019). The ability to dynamically allocate resources in response to real-time demand is a critical advantage of cloud services. This **elasticity** ensures that DevSecOps teams can handle sudden spikes in traffic or usage without manual intervention. AWS's elasticity capabilities ensure that applications remain responsive and available even during unpredictable surges (AWS n.d.).

The importance of security in DevSecOps cannot be overstated. Both the cloud provider and the user have a **shared responsibility for ensuring security**. However, cloud platforms frequently provide pre-existing security features such as encryption, identity and access management, and security monitoring tools (Saini et al. 2019). Cloud infrastructure facilitates deploying applications and services across multiple geographic regions, providing low-latency access to users around the world. AWS's global reach enables organizations to offer user-centric solutions with enhanced performance and responsiveness (AWS n.d.).

The integration of cloud infrastructure into DevSecOps practices offers numerous benefits. These advantages range from increased scalability and flexibility to enhanced automation and security measures, fundamentally transforming the operations of DevSecOps teams. However, it is crucial to strategically approach this integration by taking into account the specific needs and challenges of the organization. By leveraging the capabilities of cloud infrastructure, organizations can establish more efficient, secure, and innovative DevSecOps practices that align seamlessly with contemporary development requirements.

2.4.2 Security Considerations in Cloud Environments

Cloud computing has revolutionized the way businesses operate, providing unprecedented scalability, flexibility, and cost-efficiency (Accenture n.d.). However, along with the myriad benefits come a set of security challenges that cannot be ignored. According to **(Takabi et al., 2010)**, here are some security and privacy considerations in cloud environments...

1. **Data Protection and Privacy:** Ensuring the protection of sensitive data is of utmost importance in all computing environments, including cloud computing. Unauthorized access and breaches pose significant risks, thus organizations must utilize strong encryption methods, implement access controls, and adhere to data residency and compliance regulations to safeguard data privacy. A significant concern for cloud clients is the reluctance of numerous organizations to store their data and applica-

tions on systems that are located outside of their on-premise data centers. This fear stems from the potential risks associated with migrating workloads to a shared infrastructure, which could lead to an increased vulnerability of customers' private information to unauthorized access and exposure.

2. **Identity and Access Management (IAM):** Managing user identities and controlling access to resources are complex tasks in cloud environments. Ensuring proper authentication, authorization, and multi-factor authentication (MFA) measures are in place is crucial. IAM policies should be meticulously defined and regularly audited to prevent unauthorized access.
3. **Compliance and Regulatory Challenges:** The migration to cloud computing presents compliance and regulatory challenges that vary across industries. Organizations must verify that their cloud infrastructure meets the necessary requirements. While cloud service providers often offer compliance certifications, it remains the organization's responsibility to configure and utilize the services in a manner that complies with regulations. Cloud computing has the potential to be a worldwide phenomenon as it can utilize computing and infrastructural resources that are widely distributed. This allows cloud services to be accessible from any location and at any time. However, this can also give rise to various jurisdiction issues concerning the need for protection and enforcement mechanisms.
4. **Shared Responsibility Model:** A study by Kim et al., (2018) emphasizes the need for organizations to embrace DevSecOps to achieve a seamless integration of security throughout the software development process. This integration involves the collaboration of development, operations, and security teams, creating a shared responsibility for security across the organization. The study highlights the importance of cultural and organizational shifts in adopting DevSecOps fostering a security-first mindset among all stakeholders.

In cloud computing environments, the responsibility for security and privacy should be shared by both cloud providers and customers. However, the extent of sharing will vary depending on the delivery models chosen, which in turn impact the ability to expand cloud services. For example, in Software-as-a-Service (SaaS), providers commonly offer services with numerous integrated features, which can limit the ability of customers to customize and extend the functionality. Providers bear a higher level of responsibility for ensuring the security and privacy of application services, particularly in public cloud environments where client organizations may have strict security requirements and rely on the provider for enforcement. Private clouds, on the other hand, may necessitate greater extensibility in order to cater to specific customization needs.

2.5 Tools Selection and Rationale

There are various tools for DevSecOps security testing techniques, in this research, we only focused on the chosen tools that aligns with the implementation in this research. There are various other tools available in the market that could offer distinct insights.

2.5.1 SAST Tooling

SAST tools, possess the ability to examine not just the code that developers write and upload to their project's code repository, but also the external libraries they rely on to construct their project, especially if these libraries are open source in nature. This particular set of tasks can be effectively carried out by various software programs such as Checkmarx, Sonarqube and Sonarcloud.

SonarCloud

SonarCloud is a cloud-based service provided by SonarSource that offers static code analysis and code quality management for projects. It's designed to help developers and teams identify and address code issues, security vulnerabilities, and maintain good coding practices. Using SonarCloud, one has the capacity to analyze their codebase, receive feedback on potential issues, track code quality over time, and improve the overall reliability and security of their software projects (Cameron, 2023). To use SonarCloud, individuals typically need to integrate it with their version control system (such as GitHub, GitLab, or Bitbucket) and configure it to perform code analysis on their repositories. This integration allows SonarCloud to automatically evaluate code whenever changes are made and provide actionable insights to the user (Sonar, n.d.).

2.5.2 DAST Tooling

Web application scanners play a crucial role in identifying vulnerabilities and ensuring the security of digital systems. These tools are essential for assessing the weaknesses that could be exploited by malicious actors. They are particularly helpful because they can detect potential threats by examining how the web applications behave (Candel, 2022).

These scanners are versatile, as they can be accessed through either an API (Application Programming Interface) or a CLI (Command Line Interface). This means that users can interact with them programmatically or through command-based instructions, making them flexible and convenient to use. When it comes to performing security tests that involve dynamic analysis, there are several noteworthy open source options available.

In the study conducted by Mohan & Othmane (2016), it was demonstrated that DevSecOp is not merely a marketing term. The authors provided numerous examples to support this claim and also discussed various tools utilized in the industry to integrate security with DevOps. These tools were classified into categories such as scanning tools, security framework tools, results consolidation tools, monitoring tools, and logging tools. One specific example highlighted by the authors was the use of OWASP ZAP in scanning for vulnerabilities.

These tools are designed to scrutinize web applications while they are in action, allowing them to identify vulnerabilities that might not be apparent during static analysis. Some open source tools for DAST are OWASP ZAP, Arachni Scanner and Nikto (Rangnau et al., 2020).

OWASP ZAP

This tool stands for "Open Web Application Security Project Zed Attack Proxy" and is well-regarded for its effectiveness in dynamic security testing. It assists in revealing potential vulnerabilities by actively interacting with web applications in a manner similar to how attackers would (Daniel, 2020). OWASP ZAP functions as a tool utilized to assess the security of web applications. Its primary capabilities encompass two key aspects: conducting comprehensive scans for vulnerabilities within applications and serving as an intermediary during app usage. When scanning, it does things in three steps. First, it explores the app to find all the pages. Then, it sends test attacks to these pages. Finally, it makes a report about what it found. Custom scans can also be orchestrated through ZAP. This entails imparting specific instructions to the tool, incorporating supplementary utilities, and even invoking additional scripts. This functionality proves beneficial when necessitating more precise assessments, such as scrutinizing the security of a login system (Zap, n.d.).

2.5.3 SCA Tooling

One crucial aspect of DevSecOps is Software Composition Analysis, which involves identifying and managing open-source components and vulnerabilities within the software supply chain. Some SCA tools includes, Synk, Black Duck and WhiteSource (Snyk, 2021).

Snyk

Snyk is a developer-first security solution that focuses on identifying vulnerabilities, license issues, and open-source risks in the software dependencies used in an application. It seamlessly integrates into the development workflow, enabling developers to detect and address vulnerabilities in real-time. (Snyk 2022). Snyk doesn't just identify vulnerabilities; it also offers automated fixes and patches when available, simplifying the process of remediation. It seamlessly integrates with various Continuous Integration and Continuous Deployment (CI/CD) tools, making it possible to automate vulnerability scanning and remediation as part of the development process.

2.5.4 CI/CD Tooling

A robust Continuous Integration and Continuous Deployment (CI/CD) pipeline is pivotal for successful DevSecOps implementation. Jenkins, a widely adopted CI/CD tool, facilitates the integration of security practices at each stage of development.

Jenkins

Jenkins is an open-source automation server that facilitates the continuous integration and continuous delivery (CI/CD) of software projects (Jenkins n.d.). It was originally developed as a fork of the Hudson project in 2011 and has since become one of the most widely used tools in the software development and DevOps communities. Jenkins enables developers to integrate their code changes into a shared repository frequently, triggering automated build and test processes. This helps catch integration issues early in the development cycle. By integrating security testing tools like SonarCloud, Snyk and Zap, Jenkins enables the automated execution of security tests as part of the pipeline. This ensures that code is rigorously tested for vulnerabilities before deployment (Jenkins, n.d.).

GitLab or GitHub

GitHub or GitLab is a web-based platform that provides a collaborative environment for software developers to manage and share their code, collaborate on projects, and track changes over time. At its core, both uses a version control system called Git, which was developed by Linus Torvalds (Uzayr, 2022). Git allows developers to track changes to their codebase, collaborate with others, and maintain different versions of their projects. These platforms enhances Git's capabilities by providing a user-friendly interface, tools for collaboration, and additional features for project management (GitHub, n.d.).

One of the key features is the ability to create repositories, which are essentially folders that store your code and its entire history. Each time a change is made to the code, it's recorded in the repository's history, allowing developers to view, compare, and revert changes as needed. This fosters collaboration by enabling multiple developers to work on the same project simultaneously without interfering with each other's work (GitHub, n.d.)

2.5.5 Infrastructure as Code (IaC) Tooling

Infrastructure as Code (IaC) has become a fundamental practice in modern IT operations and software development. It's a methodology that involves using code to define and manage infrastructure resources, such as servers, networks, databases, and more. IaC tools like Terraform play a crucial role in simplifying and automating the process of provisioning and managing infrastructure.

Terraform

Terraform employs a declarative approach, where one describes the desired state of infrastructure in code, rather than scripting the exact steps to create it. This abstraction simplifies the comprehension and upkeep of infrastructure definitions, as individuals concentrate on defining the intended appearance of the infrastructure, rather than delineating the methods to attain that state. (Salecha, 2022). Terraform ensures that infrastructure setup is consistent and reproducible across different environments. One can employ the same Terraform configuration to establish identical infrastructure setups across development, testing, staging, and production environments. This reduces the chances of configuration drift and minimizes the "it works

on my machine” problem (Salecha, 2022). Infrastructure definitions written in Terraform are treated as code and can be stored in version control systems like Git. This allows teams to track changes, collaborate, and review infrastructure modifications in a controlled and systematic manner. Similar to how software code can be tracked for changes, changes to infrastructure code can also be tracked (Hashicorp n.d.).

Ibrahim et al. (2022) present a study that introduces a specialized DevSecOps module tailored for infrastructure teams. Their research focuses on the utilization of Infrastructure as Code (IaC) to address security concerns and integrate security practices into the broader DevOps cycle

2.5.6 Container Platforms

Container platforms are tools and technologies designed to manage and orchestrate the deployment, scaling, and operation of containerized applications. Containers are a form of lightweight virtualization that package an application and its dependencies, along with necessary runtime components, into a single package that can run consistently across different environments (Hsu, 2018).

Docker

Docker is one of the pioneers in the containerization space. It provides tools to create, deploy, and manage containers. Docker Engine functions as the runtime that facilitates the execution of containers. Simultaneously, Docker Compose offers the capability to define and oversee applications spanning multiple containers. Docker provides a way to package, distribute, and run applications within these containers, which include everything needed to run the application, such as code, runtime, libraries, and system tools. This eliminates the common “it works on my machine” problem by ensuring that the application runs consistently across different environments (Jangla, 2018).

A study by Zeeshan (2020) examines the unique network security considerations pertaining to Docker and containers. By familiarizing themselves with best practices for container security, developers can guarantee the integrity of their applications in containerized environments. The research also showcases demonstrations of continuous integration (CI)

processes in both on-premise and hosted environments. Furthermore, it investigates various code analysis techniques aimed at validating product quality, thereby ensuring comprehensive testing prior to deployment.

Kubernetes

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF) (Splunk, n.d.). Kubernetes provides a powerful set of tools and features to manage the complexities of deploying and scaling applications in containerized environments. Kubernetes' architecture is well-suited for microservices-based applications, which are a cornerstone of modern software development. Security considerations in microservices architecture within the DevSecOps paradigm are explored by Chandramouli (2022). The study proposes a novel threat modeling technique tailored to the unique security challenges posed by microservices-based systems. This approach effectively identifies and addresses security risks in complex architectural scenarios.

The basic deployable unit in Kubernetes. A pod can consist of one or more containers that share network and storage resources. Pods are used to encapsulate and manage containers within the same context (Splunk, n.d.).

In their study, Alawneh & Abbadi (2022) underscore the potential of DevSecOps in bolstering software security and reliability by seamlessly integrating security principles into DevOps practices. Their research delves into the challenges that organizations may encounter when implementing this approach while providing real-world examples to illustrate effective integration strategies. One key takeaway from their work is the pivotal role that DevSecOps practices play in securing complex systems, including intricate environments such as the Kubernetes ecosystem. This highlights the importance of adopting a security-centric approach in contemporary software development and operational management.

2.5.7 Cloud Environment

The cloud used to mainly help with computer infrastructure, but now it's a big part of how we make software safely and quickly. As companies want to create software faster and keep improving it, the cloud helps a lot. It makes things flexible, scalable, and automated,

which means we can add strong security measures easily. In this dynamic ecosystem, platforms like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) have emerged as powerful enablers, offering a suite of services that converge development, security, and operations into a harmonious symphony of innovation.

AWS and Its Importance

- Amazon Web Services (AWS) is a comprehensive and widely-used cloud computing platform provided by Amazon. It offers a vast array of cloud services that include computing power, storage, databases, networking, analytics, machine learning, and more. AWS enables individuals, businesses, and organizations to access and utilize computing resources without the need to invest in physical hardware or infrastructure (Mishra, 2023).
- AWS offers a wide range of security services that can be integrated into the DevSecOps process. These include Identity and Access Management (IAM), Virtual Private Cloud (VPC), Web Application Firewall (WAF), Security Hub, and more. These services enable individuals to implement a range of security controls, monitor security events, and efficiently manage access. The services offer virtual servers (instances) on which applications can be executed (AWS, n.d.). **Amazon EC2 (Elastic Compute Cloud)** is a core compute service that allows one to launch and manage instances of various types.
- It provides resizable compute capacity in the cloud, allowing users to launch and manage virtual servers, commonly known as instances. EC2 instances are designed to run a variety of workloads, from simple web applications to complex, high-performance computing tasks (AWS n.d.). EC2 instances have the capability to be initiated within the confines of an Amazon Virtual Private Cloud (VPC), affording the user the ability to outline private subnets, security groups, and access control lists. This configuration facilitates the isolation and security of said instances. Additionally, Elastic IP addresses can be allocated to instances to enable the allocation of static public IP addresses (AWS, n.d.).

Amazon Elastic Kubernetes Service (Amazon EKS)

Amazon EKS is a **managed Kubernetes** service provided by Amazon Web Services (AWS). Kubernetes is an open-source container orchestration platform that helps automate the deployment, scaling, and management of containerized applications. Amazon EKS simplifies the process of running and managing Kubernetes clusters on AWS infrastructure. Amazon EKS takes care of the heavy lifting involved in setting

up, operating, and scaling Kubernetes clusters. This includes managing the control plane, networking, and worker nodes, allowing developers to focus on deploying applications, implement security controls using network policies and pod security policies to protect applications and data (Ifrah, 2019).

Amazon Elastic Container Registry (ECR)

Amazon ECR is a fully managed container registry service provided by Amazon Web Services (AWS). It's designed to store, manage, and deploy Docker container images and other container artifacts. ECR seamlessly integrates with other AWS services like Amazon ECS (Elastic Container Service) and Kubernetes, making it easier to develop, build, and deploy containerized applications. ECR provides the capability for the storage and management of Docker container images within private repositories. This guarantees the secure retention of container images, granting access solely to authorized users and services. It also offers image scanning capabilities to help identify and remediate security vulnerabilities and software package issues in container images. This can enhance the security of containerized applications (AWS, n.d.).



Figure 2.4: Image showing some services offered by AWS (Logz n.d.)

2.5.8 Infrastructure Setup

The AWS cloud environment is chosen for the implementation of the DevSecOps pipeline. Infrastructure setup is achieved using Infrastructure as Code (IaC) principles through Terraform. AWS Kubernetes is employed to manage container orchestration, enabling scalability and efficient resource utilization. The implementation infrastructure consists of virtual machines, container clusters, and networking components. These components are provisioned and configured according to best practices for security and performance. Automation ensures consistent infrastructure setup and minimizes configuration errors.

Through the seamless integration of security practices into each stage of the development lifecycle, the assurance is established that our application not only fulfills functional requisites but also remains resilient in the face of the dynamic threat landscape. This approach provides users with a dependable and secure digital experience.

2.6 Research Gaps and Challenges

While previous research has explored individual security testing methodologies, such as SAST or DAST, there is a notable gap in the literature regarding the comprehensive integration of multiple methodologies within a DevSecOps pipeline (Rangnau et al. 2020). This research aims to bridge this gap by proposing a holistic approach that synergistically combines SAST, SCA, and DAST, leading to a more robust and encompassing security assessment. Although there is an abundance of theoretical discussions on DevSecOps and security testing, there is a lack of practical implementation guidelines (Myrbakken & Colomo-Palacios 2017). This research aims to fill this gap by offering a step-by-step guide to establish a DevSecOps pipeline, incorporate security tools, and deploy applications in a cloud environment. By providing a hands-on perspective, this study contributes to the practical application of theoretical concepts.

One of the primary challenges encountered during this research was ensuring the seamless integration and compatibility of various security testing tools within the DevSecOps pipeline. While each tool individually proved effective, configuring them to work harmoniously required meticulous effort and overcoming interoperability challenges. Another challenge emerged in the form of false positives and negatives generated by the security testing tools. SAST and DAST tools occasionally flagged issues that required manual re-

view to ascertain their validity. This challenge underscores the need for continuous tool refinement and emphasizes the importance of human intervention in the assessment process. The research focused on AWS for cloud infrastructure, yet cloud environments offered by different providers have distinct characteristics and security considerations. This posed a challenge in ensuring the adaptability of the proposed methodology to various cloud platforms, requiring careful consideration of platform-specific intricacies.

2.7 Conclusion

This chapter has looked closely at the main ideas behind DevSecOps, the problems we face in securing software, and the different ways we test for security. We've also seen how important it is to bring security into the whole development process. In the next chapters, we'll go deeper into actually doing DevSecOps. We'll see how to put tools like SAST, SCA, and DAST into a pipeline that uses Jenkins. We'll also set up the needed foundation using Terraform and Kubernetes on AWS. By doing this, we'll understand better how DevSecOps makes software more secure. The knowledge distilled from this chapter serves as the bedrock upon which the forthcoming chapters will build, facilitating a deeper comprehension of the pragmatic integration of DevSecOps practices. By harmonizing theoretical insights with practical execution, this research aspires to not solely enrich the theoretical dialogue encompassing DevSecOps but also to directly empower its tangible implementation. Through this synergy, the study aims to make tangible contributions to the safeguarding of software applications within the ever-evolving digital panorama.

Chapter 3

METHODOLOGY

3.1 Research Design and Approach

To address the research objectives effectively, an experimental research design was chosen. This approach is particularly suitable for evaluating the impact of security testing methodologies on software security within the DevSecOps pipeline. The experimental approach involves a case study where a DevSecOps pipeline is constructed and security testing methodologies are integrated. Section 3.2 below shows the end-end case study of how the security tools will be implemented. In order to Implement this DevSecOps I will be using a buggy Java web application that will be deployed on AWS.

3.2 End to End Case Study That Will Be Implemented

A comprehensive step-by-step implementation guide to establish a DevSecOps pipeline for deploying a secure web application on AWS is shown in the steps below. These steps covers, a comprehensive DevSecOps pipeline that covers static and dynamic security testing, as well as software composition analysis, for web application deployed on AWS resources.

Step 1 - Create AWS EC2 Instance with Jenkins and Other Software using Terraform:

The Implementation commences with the foundational step of provisioning an AWS EC2 instance through Terraform. This automated infrastructure as code (IaC) approach not only accelerates environment setup but also ensures consistency across deployments.

On this EC2 instance, we will install Jenkins, a leading automation server, and other essential software components.

Step 2 - Configure Jenkins with Kubernetes, ECR, AWS Plugins, and Credentials

With our EC2 instance up and running, we delve into configuring Jenkins to seamlessly integrate with AWS services and Kubernetes. By installing important plugins and setting up AWS and Kubernetes credentials, this empowers Jenkins to orchestrate the deployment process.

Step 3 - Integrate SonarCloud for SAST in DevSecOps Pipeline

By integrating SonarCloud into our pipeline, we integrate a robust static application security testing solution. SonarCloud meticulously scrutinizes the codebase, identifying vulnerabilities and enforcing coding best practices. This proactive approach empowers developers to rectify security issues during the development phase itself.

Step 4 - Integrate Snyk for SCA in DevSecOps Pipeline

To comprehensively safeguard our application, we introduce Snyk for software composition analysis. By scanning and assessing third-party dependencies for vulnerabilities, we mitigate potential risks originating from open-source components. This integration ensures that our application is built upon a secure foundation.

Step 5 - Build and Push the Buggy Web Application to AWS ECR

The process of building and containerizing the application is essential for deployment scalability and consistency. Through our pipeline, we automate the building of Docker images, ensuring uniformity across various environments. These images are then securely pushed to Amazon Elastic Container Registry (ECR), a fully managed Docker container registry, facilitating version control and artifact management.

Step 6 - Deploy the Web Application from Amazon ECR to AWS EKS

Leveraging the power of Kubernetes, we smoothly deploy our containerized application onto an Amazon Elastic Kubernetes Service (EKS) cluster. Kubernetes orchestrates container deployment, scaling, and management, ensuring optimal application performance and resource utilization.

Step 7 - Run DAST Scan on the Deployed Web Application

Our pipeline doesn't stop at deployment; it continues with dynamic application security testing (DAST) powered by OWASP ZAP. ZAP rigorously tests our deployed application for vulnerabilities and generates detailed reports, enabling proactive identification and remediation of security gaps.

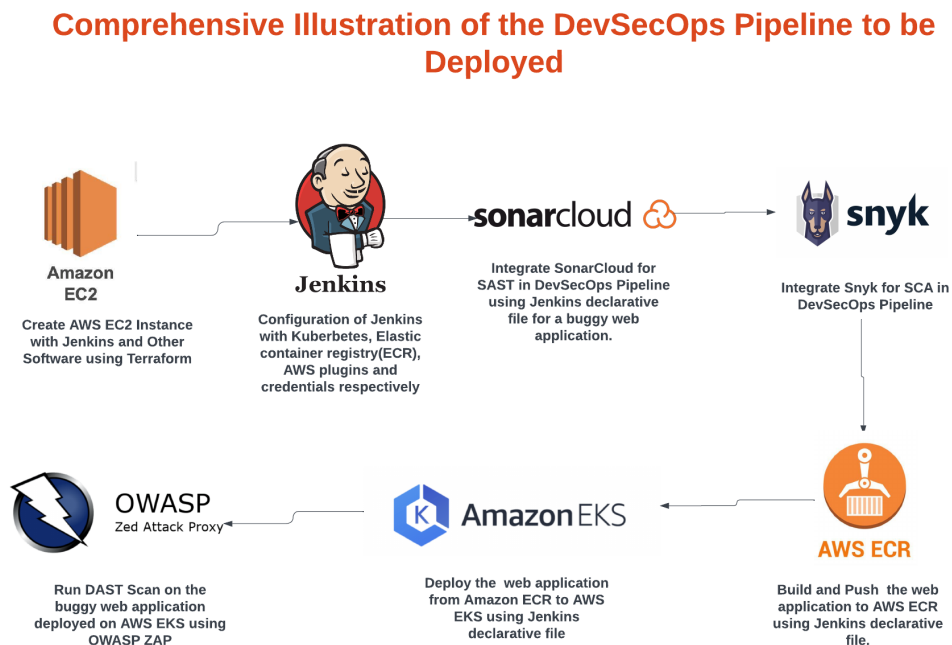
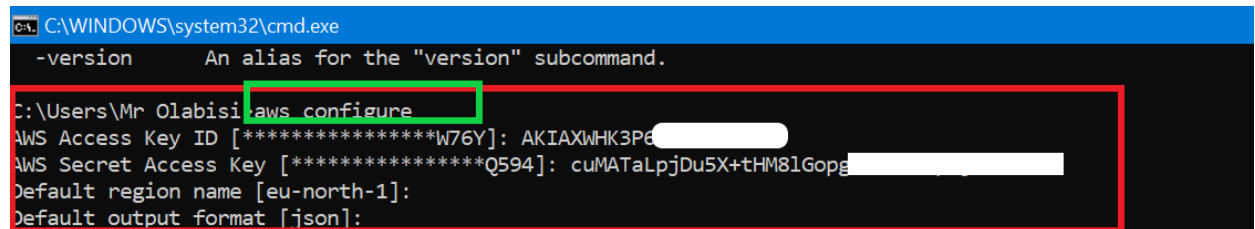


Figure 3.1: Comprehensive Illustration of the DevSecOps Pipeline to be Deployed

3.3 Cloud Infrastructure

The process of implementing the DevSecOps pipeline starts by creating a strong infrastructure capable of coordinating security practices and development workflows. Amazon Web Services (AWS) is a suitable platform for this purpose due to its scalability and flexibility. The first step was to create an account on AWS. After that, an IAM (Identity and Access Management) user with admin privileges was created in order to have full access to all AWS services and resources within the account (AWS, n.d.). This level of access allows the user to manage, create, modify, and delete any AWS resources, including launching and terminating instances(virtual machines), managing storage, configuring networking,

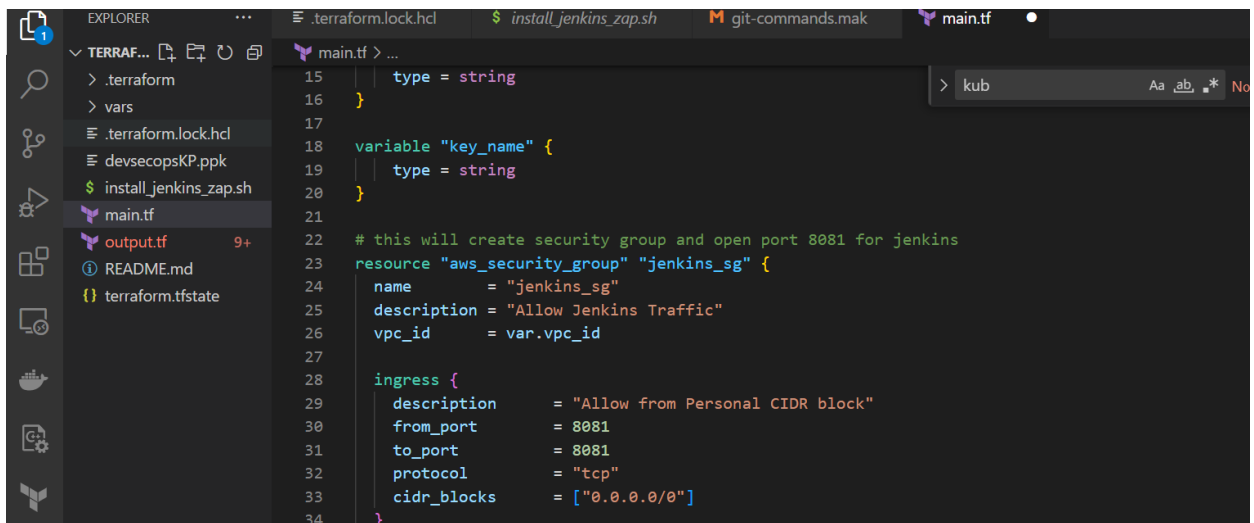
and much more. Using IAM users with restricted permissions instead of the root account is a recommended best practice for managing access to AWS resources. Using the root account for day-to-day operations is discouraged due to security and operational concerns. The AWS Command Line Interface (CLI) was installed and configured as an essential component of our preliminary setup. The CLI provides a powerful command-line interface for interacting with AWS services and resources. By configuring the CLI with the IAM user credentials we created, we ensure a streamlined and programmatic way to manage AWS resources, execute scripts, and automate tasks as shown below (AWS, n.d.).



```
C:\WINDOWS\system32\cmd.exe
-aws configure
An alias for the "version" subcommand.
C:\Users\Mr Olabisi>aws configure
AWS Access Key ID [*****W76Y]: AKIAWHK3P6
AWS Secret Access Key [*****Q594]: cuMATaLpjDu5X+tHM8lGop
Default region name [eu-north-1]:
Default output format [json]:
```

Figure 3.2: Configuring the AWS CLI for the IAM user for programmatic access

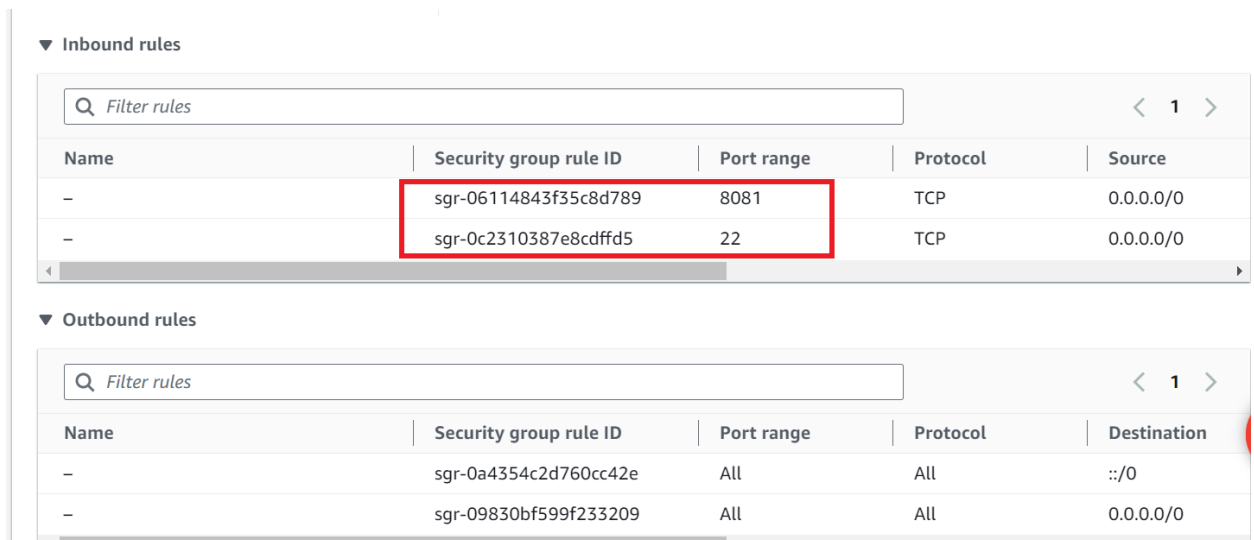
The process commences with defining the infrastructure components required for the DevSecOps pipeline. These components include an AWS EC2 instance(Virtual Machine) and the necessary networking configurations like Security Groups. Security group act as virtual firewalls that control inbound and outbound traffic to EC2 instances and other AWS resources. Security Groups define rules that determine which types of traffic are allowed and denied, based on IP addresses, ports, and protocols using terraform configuration files, written in HashiCorp Configuration Language (HCL), articulate the desired state of the infrastructure.

A screenshot of a code editor with a dark theme. The left sidebar shows a file explorer with folders like .terraform, vars, and files like .terraform.lock.hcl, devsecopsKP.ppk, install_jenkins_zap.sh, main.tf, output.tf, README.md, and terraform.tfstate. The main editor area shows the content of main.tf. The code defines a variable 'key_name' of type string, a comment about creating a security group, and an 'aws_security_group' resource named 'jenkins_sg'. The resource has attributes for name, description, vpc_id, and an ingress rule allowing TCP traffic on port 8081 from the CIDR block 0.0.0.0/0.

```
15 | type = string
16 | }
17 |
18 | variable "key_name" {
19 |   type = string
20 | }
21 |
22 | # this will create security group and open port 8081 for jenkins
23 | resource "aws_security_group" "jenkins_sg" {
24 |   name           = "jenkins_sg"
25 |   description    = "Allow Jenkins Traffic"
26 |   vpc_id         = var.vpc_id
27 |
28 |   ingress {
29 |     description = "Allow from Personal CIDR block"
30 |     from_port   = 8081
31 |     to_port     = 8081
32 |     protocol    = "tcp"
33 |     cidr_blocks = ["0.0.0.0/0"]
34 |   }
}
```

Figure 3.3: The Terraform main.tf code snippet written in HashiCorp Configuration Language (HCL)

In this case, inbound traffic to specific ports, such as port 8081 for Jenkins and port 22 was allowed. By allowing inbound traffic on port 8081 in our AWS security group, we enabled external connections to reach the Jenkins server running within our AWS infrastructure. This is essential for tasks like accessing the Jenkins web interface, interacting with Jenkins jobs, and monitoring the progress of CI/CD pipelines and by allowing inbound traffic on port 22, we enabled users or systems to establish SSH connections to our AWS instances.

A screenshot of the AWS Management Console showing the 'Inbound rules' and 'Outbound rules' for a security group. The 'Inbound rules' table has two rules highlighted with a red box: one for port 8081 (TCP) and one for port 22 (TCP). The 'Outbound rules' table shows two rules for all ports (All) using the protocol 'All'.

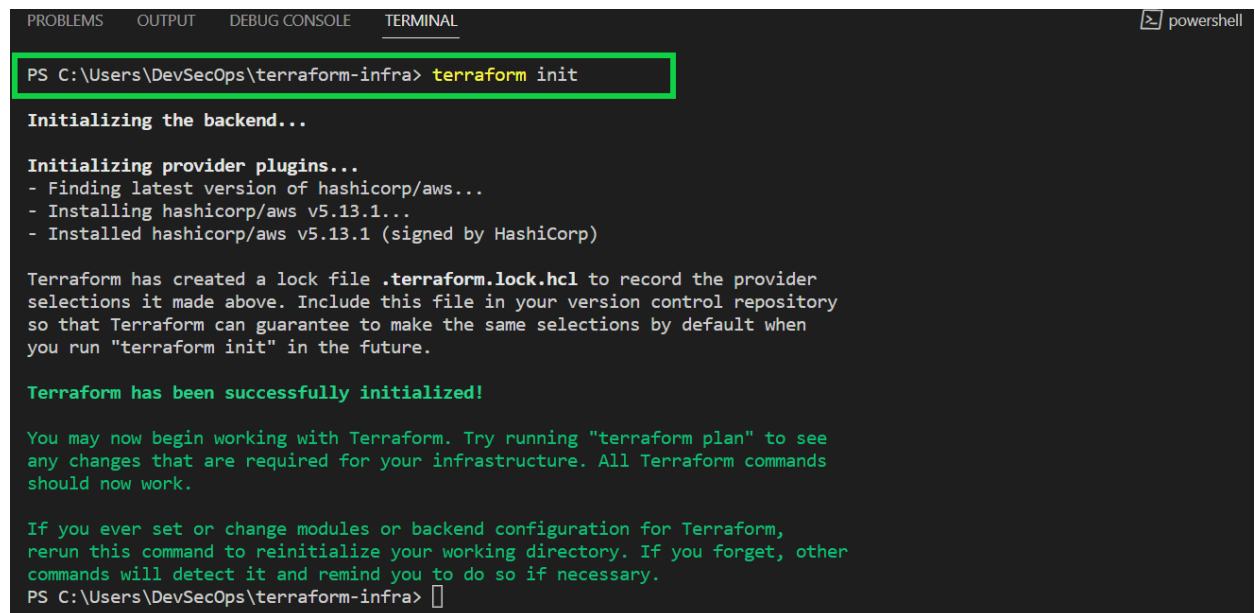
Inbound rules				
Name	Security group rule ID	Port range	Protocol	Source
-	sgr-06114843f35c8d789	8081	TCP	0.0.0.0/0
-	sgr-0c2310387e8cdffd5	22	TCP	0.0.0.0/0

Outbound rules				
Name	Security group rule ID	Port range	Protocol	Destination
-	sgr-0a4354c2d760cc42e	All	All	::/0
-	sgr-09830bf599f233209	All	All	0.0.0.0/0

Figure 3.4: Image showing the ports opened to allow traffic as port 8081 is for jenkins and 22 for SSH

Terraform's declarative nature allows developers to specify the desired resources and their

interdependencies. Through a series of commands, such as **terraform init**, **terraform plan**, and finally **terraform apply**, the infrastructure is created in accordance with the defined configuration in the **main.tf** of the terraform repo. This not only ensures consistency across environments but also provides an audit trail of changes made to the infrastructure.



```
PS C:\Users\DevSecOps\terraform-infra> terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.13.1...
- Installed hashicorp/aws v5.13.1 (signed by HashiCorp)

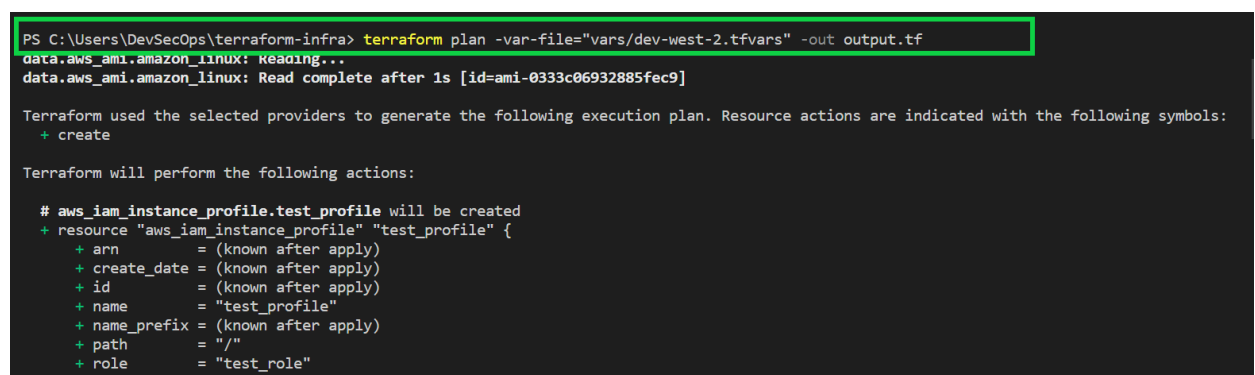
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
PS C:\Users\DevSecOps\terraform-infra>
```

Figure 3.5: Using terraform init to initialize a terraform Working Directory



```
PS C:\Users\DevSecOps\terraform-infra> terraform plan -var-file="vars/dev-west-2.tfvars" -out output.tf
data.aws_ami.amazon_linux: Reading...
data.aws_ami.amazon_linux: Read complete after 1s [id=ami-0333c06932885fec9]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_iam_instance_profile.test_profile will be created
+ resource "aws_iam_instance_profile" "test_profile" {
  + arn          = (known after apply)
  + create_date  = (known after apply)
  + id           = (known after apply)
  + name         = "test_profile"
  + name_prefix  = (known after apply)
  + path         = "/"
  + role         = "test_role"
  + tags_all     = (known after apply)
}
```

Figure 3.6: Image showing the resources that will be created within AWS using terraform plan

```

PS C:\Users\DevSecOps\terraform-infra> terraform apply "output.tf"
aws_iam_role.test_role: Creating...
aws_security_group.jenkins_sg: Creating...
aws_iam_role.test_role: Creation complete after 1s [id=test_role]
aws_iam_role_policy.test_policy: Creating...
aws_iam_instance_profile.test_profile: Creating...
aws_iam_role_policy.test_policy: Creation complete after 1s [id=test_role:test_policy]
aws_iam_instance_profile.test_profile: Creation complete after 1s [id=test_profile]
aws_security_group.jenkins_sg: Creation complete after 2s [id=sg-00550ce2fb40c2e03]
aws_instance.web: Creating...
aws_instance.web: Still creating... [10s elapsed]
aws_instance.web: Still creating... [20s elapsed]
aws_instance.web: Still creating... [30s elapsed]
aws_instance.web: Still creating... [40s elapsed]
aws_instance.web: Creation complete after 41s [id=i-05695e883435e9d2f]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
PS C:\Users\DevSecOps\terraform-infra>

```

Figure 3.7: Creating the aws resources as declared in the terraform main.tf configuration file using terraform apply

In the Terraform repo was a shell scripts called **installjenkinszap.sh**. The shell script is a sequence of commands written in a scripting language that automates the installation and configuration of various software packages and tools on the virtual machine. The shell scripts is to install Jenkins, OWASP ZAP (a security tool), Maven (a build tool), Docker, Kubernetes, and Git. By including the shell script execution in the Terraform configuration main.tf file, It ensures that these software's components are installed and configured on the EC2 instance as soon as it's provisioned. This seamless integration automates the setup process, saving time and effort and ensuring consistency across environments.

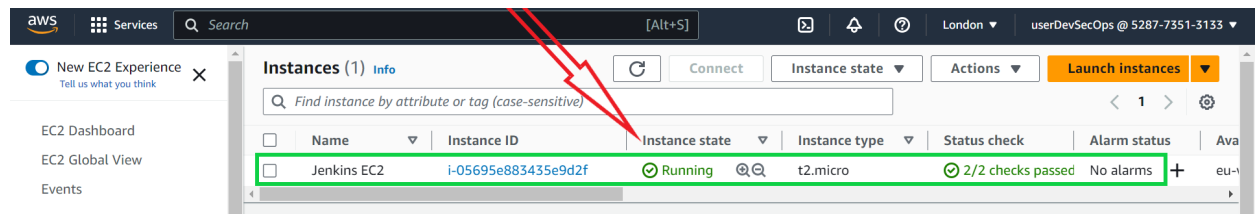


Figure 3.8: Running virtual machine created on AWS with Jenkins, Kubernetes, Docker OWASP ZAP and Maven installed

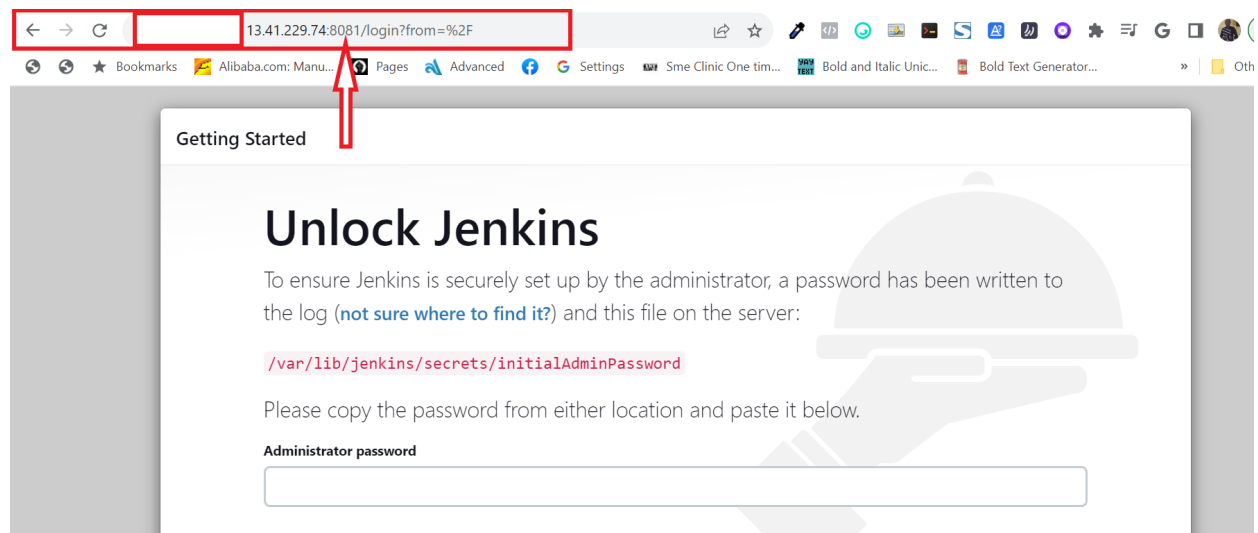


Figure 3.9: Jenkins successfully installed and accessed through port 8081

3.4 CI/CD Software

3.4.1 Jenkins and Its Plugins

Once the foundational infrastructure is in place, the journey towards integrating security testing into the Software Development Lifecycle (SDLC) continues with the configuration of Jenkins and the seamless integration of essential plugins. Jenkins, an automation server renowned for its extensibility and versatility, plays a pivotal role in orchestrating the DevSecOps pipeline. This section delves into the intricacies of configuring Jenkins, integrating relevant plugins, and establishing the framework for secure and continuous software development. These plugins enhance Jenkins' capabilities, enabling it to interface seamlessly with AWS services, Docker containers, and Kubernetes clusters.

The plugins installed for this implementation are CloudBees AWS credentials, Docker Pipeline, AWS ECR and Kubernetes CLI

CloudBees AWS credentials: This plugin acts as a bridge between Jenkins and AWS services. It enables the secure storage and management of AWS credentials within Jenkins, ensuring that sensitive information such as access keys and tokens are kept encrypted and well-managed. This integration streamlines interactions between Jenkins and AWS services, such as Elastic Container Registry (ECR) and Elastic Kubernetes Service (EKS).

Docker Pipeline Plugin: Containers have revolutionized software deployment by encapsulating applications and their dependencies. The Docker Pipeline plugin enables Jenkins to interact with Docker containers, facilitating the building, testing, and deployment of applications within containers.

AWS ECR Plugin: As a fundamental component of containerization, the AWS Elastic Container Registry (ECR) plugin empowers Jenkins to interact with the ECR service. This integration streamlines the process of pushing and pulling Docker images to and from the ECR repository, enabling the seamless distribution of containerized applications.

Kubernetes CLI Plugin: The Kubernetes CLI plugin enables Jenkins to interact with Kubernetes clusters, facilitating the deployment and management of applications on Kubernetes. This integration aligns with the goal of achieving automated and efficient application deployment.

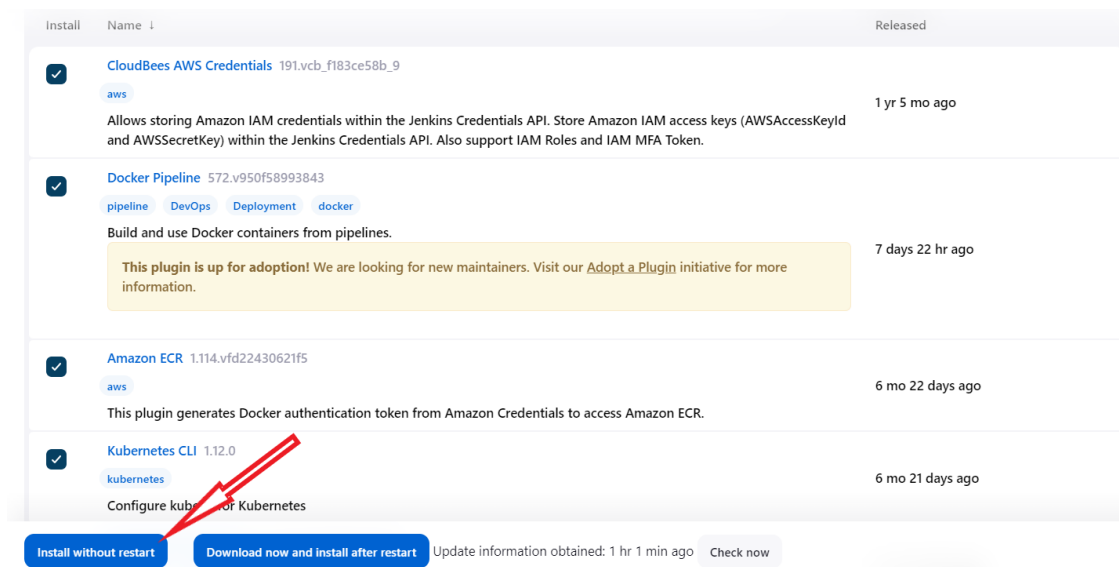


Figure 3.10: Installation of important plugins on Jenkins

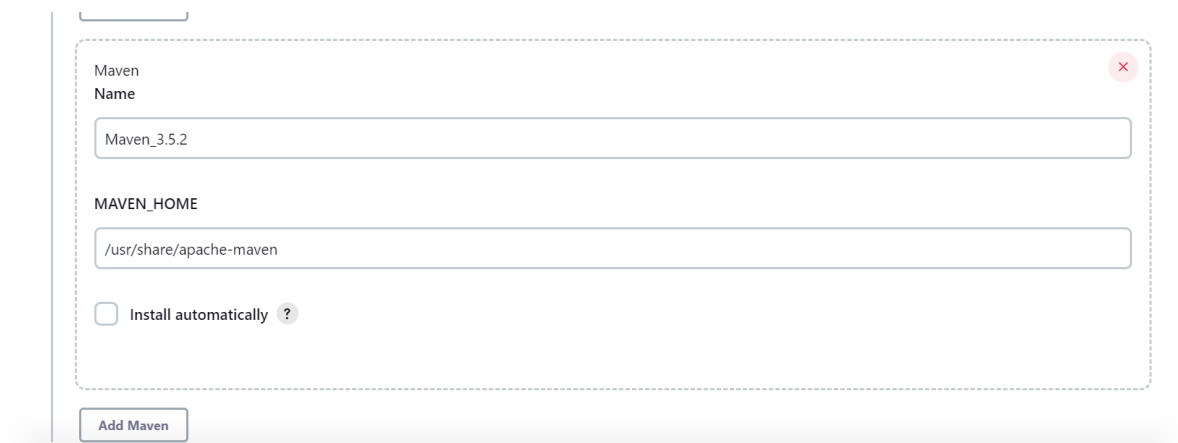
3.4.2 Project Compilation With Maven

The chosen application for testing is developed using Java programming language and employs Apache Maven as its build automation and project management tool. Maven simplifies the compilation and management of the project by reading configuration information from XML files, commonly referred to as "pom.xml" files. These files contain essential details necessary for generating the application's executable code. The pom.xml specifies

dependencies, build settings, and profiles for different JDK versions. It also integrates various plugins for tasks such as compiling, packaging, running, and scanning the application. The Snyk plugin is used to perform Software Composition Analysis to identify and manage vulnerabilities in the project's dependencies.

```
[ec2-user@ip-172-31-35-179 ~]$ mvn --version
Apache Maven 3.5.2 (138edd61fd100ec658bfa2d307c43b76940a5d7d; 2017-10-18T07:58:13Z)
Maven home: /usr/share/apache-maven
Java version: 17.0.6, vendor: Amazon.com Inc.
Java home: /usr/lib/jvm/java-17-amazon-corretto.x86_64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.14.320-243.544.amzn2.x86_64", arch: "amd64", family: "unix"
[ec2-user@ip-172-31-35-179 ~]$
```

Figure 3.11: Image showing maven version 3.5.2 installed on the virtual machine



The image shows a Jenkins configuration window for the Maven plugin. It has a title bar with a close button. Inside, there's a 'Name' label with a text input field containing 'Maven_3.5.2'. Below that is a 'MAVEN_HOME' label with a text input field containing '/usr/share/apache-maven'. At the bottom left, there's an unchecked checkbox labeled 'Install automatically' with a help icon. At the bottom center, there's an 'Add Maven' button.

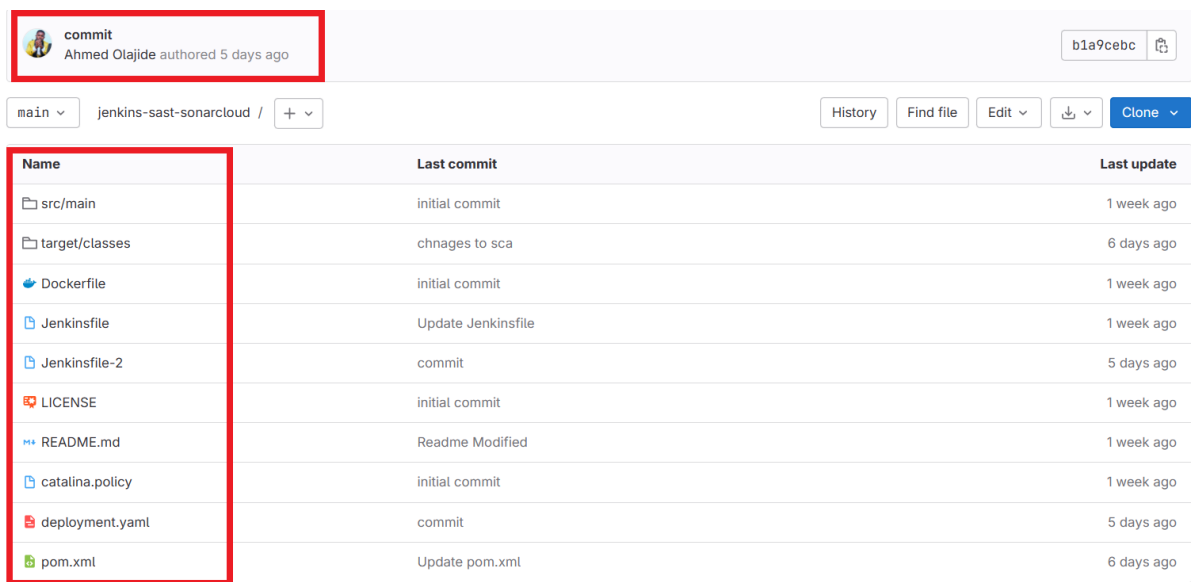
Figure 3.12: Maven plugin being configured on Jenkins with the maven home directory

3.4.3 Git Repository

The actual source code of the web application use is stored in repositories on GitLab. Each repository contains all the files and directories that make up the application, including the Java source files, configuration files, and the all-important pom.xml file that Maven uses for build instructions. The Jenkins server is configured to monitor the GitLab repository for any changes. This is done using a webhook, which is a mechanism that notifies Jenkins whenever a new commit is made to the repository. When a new commit is detected on the GitLab repository, Jenkins triggers a process to "pull" the latest version of the source code from the repository. This ensures that the Jenkins server has the most up-to-date code to work with. Once the source code is obtained, Jenkins uses the Maven build tool to execute the build process. This involves compiling the Java code, resolving dependencies, and

generating executable artifacts. The instructions for this build process are specified in the project's pom.xml file.

As mentioned earlier, the project's pom.xml file also includes configurations for various Maven plugins. For instance, the Snyk and OWASP's dependency-check plugins which are used to perform code analysis and security checks. Jenkins ensures that these plugins are executed as part of the build process. As the build process proceeds, Jenkins captures information about the build, such as any errors or warnings that occur during compilation or analysis.



Name	Last commit	Last update
src/main	initial commit	1 week ago
target/classes	chnages to sca	6 days ago
Dockerfile	initial commit	1 week ago
Jenkinsfile	Update Jenkinsfile	1 week ago
Jenkinsfile-2	commit	5 days ago
LICENSE	initial commit	1 week ago
README.md	Readme Modified	1 week ago
catalina.policy	initial commit	1 week ago
deployment.yaml	commit	5 days ago
pom.xml	Update pom.xml	6 days ago

Figure 3.13: Image showing the hosted java application on GitLab repository where jenkins pulls from

Jenkinsfile

In the GitLab repository is also a **"Jenkinsfile"** with instructions to automates the entire build, test, and code analysis process for the Java web app. It leverages Maven for project management and execution, as well as the SonarCloud plugin for static code analysis. The pipeline defines a single stage named CompileandRunSonarAnalysis, which involves cleaning, compiling, testing the project, and conducting a SonarQube analysis. The SonarCloud analysis results are then sent to SonarCloud for comprehensive code quality assessment. This pipeline structure enhances the development process by ensuring code integrity, continuous integration, and monitoring of code quality metrics.

3.4.4 Container Image Compilation With Docker

Compiling a container image using Docker involves creating a "**Dockerfile**" that outlines the necessary steps to build an image containing the application and its dependencies. A Docker image is a lightweight, standalone, and executable package that encapsulates an application along with its runtime environment, libraries, and dependencies. Image creation is a crucial step in the containerization process, as it ensures that the application runs consistently across different environments, regardless of variations in the underlying infrastructure. In this case, The Dockerfile sets up a multi-stage build process. In the first stage, it builds the Java application using Maven. In the second stage, it creates a runtime environment using OpenJDK 8 and runs the Java application with a variety of runtime arguments and configurations. The resulting Docker image will be an isolated environment containing the web application, ready to be run as a container.



Figure 3.14: Image of the dockerfile using the official Maven 3.8 image with JDK 8 to build our application

3.4.5 Kubernetes Orchestration Integration

A Kubernetes cluster is comprised of nodes, which are virtual instances on AWS. Nodes are the computing resources that run containerized applications. Each cluster has a control plane, responsible for managing the overall cluster and making global decisions about the state of the system. The smallest deployable unit in Kubernetes is a pod. A pod encapsulates one or more containers, sharing the same network namespace and storage. This allows related containers to communicate easily and ensures that they are co-located on the same node.

```

ec2-user@ip-172-31-38-110 ~]$ eksctl create cluster --name kubernetes-cluster --version 1.23 --region us-west-2 --nodegroup-name linux-nodes --n
e-type t2.xlarge --nodes 2
2023-08-23 20:12:21 [i] eksctl version 0.153.0
2023-08-23 20:12:21 [i] using region us-west-2
2023-08-23 20:12:22 [i] skipping us-west-2d from selection because it doesn't support the following instance type(s): t2.xlarge
2023-08-23 20:12:22 [i] setting availability zones to [us-west-2b us-west-2a us-west-2c]
2023-08-23 20:12:22 [i] subnets for us-west-2b - public:192.168.0.0/19 private:192.168.96.0/19
2023-08-23 20:12:22 [i] subnets for us-west-2a - public:192.168.32.0/19 private:192.168.128.0/19
2023-08-23 20:12:22 [i] subnets for us-west-2c - public:192.168.64.0/19 private:192.168.160.0/19

```

Figure 3.15: Creating a Kubernetes Cluster on AWS

From the figure above, an Amazon EKS cluster is being created in the EU (London) region (eu-west-2) with the name "kubernetes-cluster" and utilizing Kubernetes version 1.23. A node group named "linux-nodes" is being configured, consisting of EC2 instances of the "t2.xlarge" type and a total of 2 nodes. The purpose of this cluster is to manage and orchestrate containerized applications using Kubernetes.

Deployment.yaml File: The deployment.yaml file located in the GitLab repository serves as a means to specify a deployment resource in Kubernetes. This resource functions as a higher-level abstraction that facilitates declarative updates to applications. Essentially, a Deployment in Kubernetes is responsible for overseeing the lifecycle of a group of pods, simplifying the management of application updates, scaling, and rollbacks. By utilizing a deployment.yaml file, we define how our application should be deployed and supervised by Kubernetes. Upon applying this YAML file using the "kubectl apply -f deployment.yaml" command, Kubernetes generates a Deployment resource based on the provided specifications as shown in the code snippet below.

```

! deployment.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: easybuggy-deployment
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: easybuggy
10   template:
11     metadata:
12       labels:
13         app: easybuggy
14     spec:
15       containers:
16       - name: easybuggy
17         image: 528773513133.dkr.ecr.eu-west-2.amazonaws.com/easy
18         imagePullPolicy: Always

```

Figure 3.16: Deployment.yaml Code Snippet

3.5 Conclusion

This chapter has outlined how the research was conducted to enhance security practices in the DevSecOps pipeline and establish a cloud infrastructure. The study used a hands-on approach to gather real-world insights into how security can be seamlessly integrated into the software development process. The next chapter will provide step-by-step details of how these security tools were put into action, including building the DevSecOps pipeline and incorporating security tools. This thorough exploration will help us better grasp how adopting DevSecOps practices can influence and improve software security.

Chapter 4

IMPLEMENTATION AND RESULTS

This chapter embarks on the practical journey of implementing the DevSecOps pipeline with integrated security testing methodologies. The implementation follows the experimental design outlined in Chapter 3. The chapter also presents the outcomes of the implementation, including the identification of vulnerabilities, challenges faced, and insights gained from the practical integration of security into the software development lifecycle.

4.1 Target Web Application

The target web application used for this research is an already vulnerable application called **EasyBuggy** (k-tamura n.d.). By intentionally introducing bugs, vulnerabilities, and weaknesses, EasyBuggy provides a hands-on platform for individuals to explore and study real-world scenarios in a controlled environment. Users can experiment with the application to discover the consequences of different kinds of coding mistakes, security lapses, and misconfigurations that might occur in a real web application. This can include issues like cross-site scripting (XSS), SQL injection, cross-site request forgery (CSRF), and more (k-tamura n.d.).

4.2 SAST Implementation

When integrating SonarCloud into the pipeline, several steps are taken to ensure its effective operation. During the pipeline's build process, the source code is sent to SonarCloud for analysis. The tool scans the codebase, looking for patterns and signatures of known vulnerabilities and quality issues. SonarCloud identifies vulnerabilities, such as insecure

coding practices, improper input validation, and potential entry points for attackers. It also detects code quality issues that could impact the application's overall security and maintainability.

```
pipeline {
  agent any
  tools {
    maven 'Maven_3_5_2'
  }
  environment {
    SONAR_TOKEN = credentials('SONAR_TOKEN')
  }

  //SONAR ANALYSIS FOR SAST SCAN
  stages{
    stage('CompileandRunSonarAnalysis') {
      steps {
        sh 'mvn clean verify sonar:sonar -Dsonar.projectKey=devsecopsjenkins -Dsonar.organization=devsecopsjenkins -Dsonar.host.url=https://sonarcloud.io -Dsonar.token=${SONAR_TOKEN}'
      }
    }
  }
}
```

Figure 4.1: SonarCloud Pipeline Stage Implementation In Jenkinsfile

4.2.1 Parameters for SonarCloud Configuration

In the configuration segment show in figure 4.1 above, we establish the framework for the entire pipeline execution. It designates that the pipeline can be executed on any available agent (worker machine) through the **"agent any"** directive. Additionally, the **"tools"** section designates the usage of Maven version **"Maven_3_5_2"** for the build processes. Furthermore, an environment variable **"SONAR_TOKEN"** is defined in the **"environment"** block, intended to store the SonarCloud token retrieved from the Jenkins credentials. The **Stage** step accomplishes the tasks of compiling the source code, conducting static code analysis using SonarQube, and subsequently forwarding the analysis findings to SonarCloud.

4.2.2 SonarCloud Build and Test Result

The result of the SAST build and testing result are shown in the figures below respectively.

Stage View

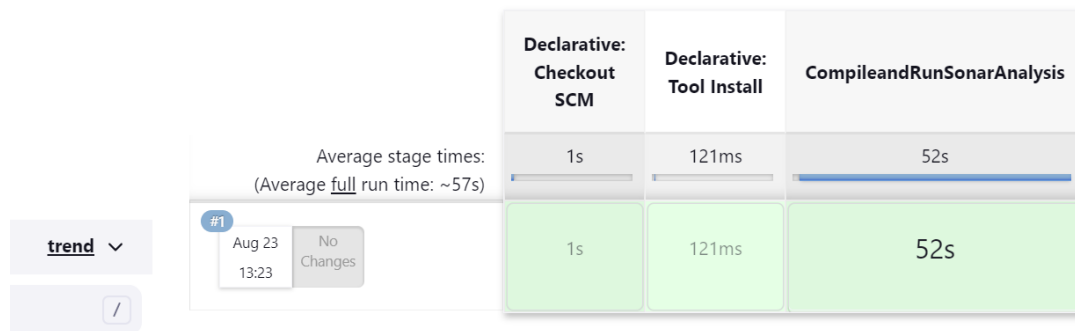


Figure 4.2: SonarCloud Pipeline Build Result In Jenkins

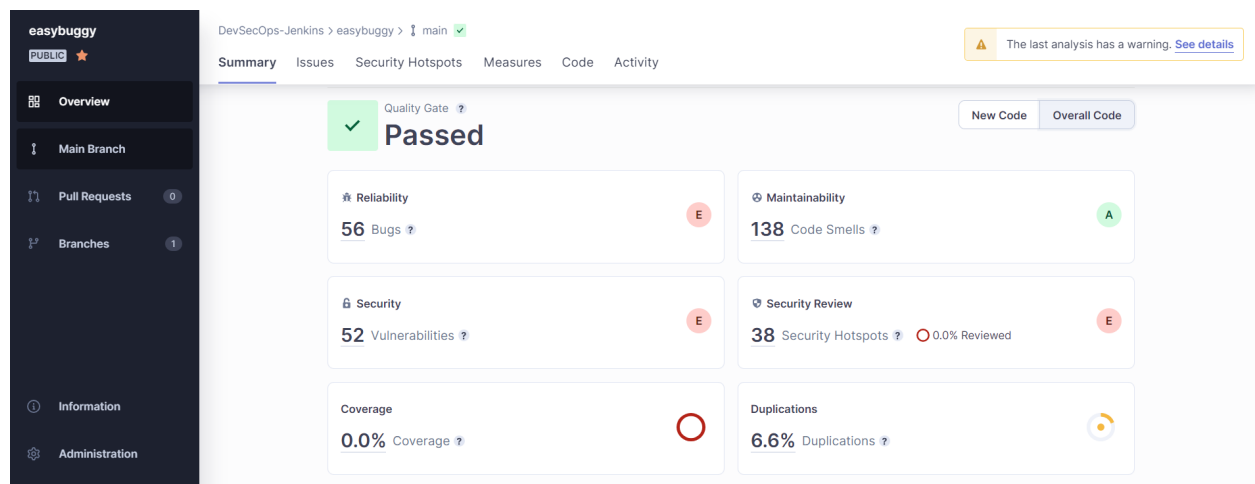


Figure 4.3: SonarCloud Vulnerability Scan Result

After the build process has been completed by Jenkins, it sends the code to SonarCloud for SAST analysis. SonarCloud scan result shows we have 56 bugs, 53 vulnerabilities, 138 maintainability code smells that is confusing and difficult to maintain, 38 security hotspot which are security-sensitive code that requires manual review to assess whether or not a vulnerability exists and 6.6% Identical lines of code. Not only does it scan for issues in the code, it also provides solution on what one could do to prevent this issues from the code. The figure below shows an example of a security issue found and what is needed to be done to solve it.

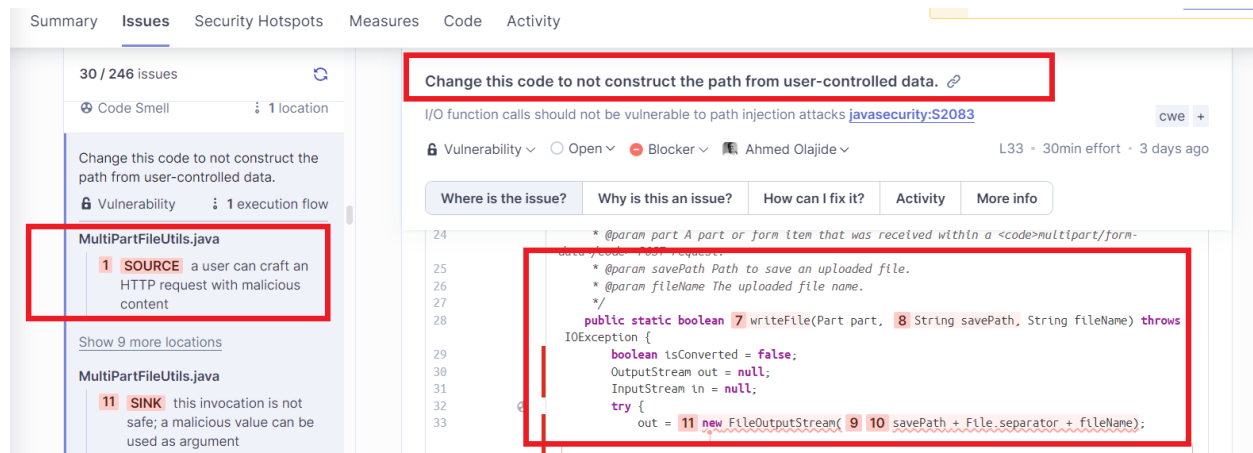


Figure 4.4: Image Showing SonarCloud Recommended Solution To Fix the Code Issue

4.3 SCA Implementation

The Purpose of performing SCA scan in our code is to check for security vulnerabilities in dependencies since most of our codes in the pom.xml file relies on third-party libraries, frameworks, and components. With Snyk integration into the development process, it provides us with vulnerabilities in the dependencies used in the code.

Below is the pipeline stage added to the Jenkinsfile to perform a SCA scan using Snyk.

```

17
18 //SYNK ANALYSIS FOR SCA SCAN
19 stage('RunSCAAnalysisUsingSnyk') {
20     steps {
21         withCredentials([string(credentialsId: 'SNYK_TOKEN', variable: 'SNYK_TOKEN')]) {
22             sh 'mvn snyk:test -fn'
23         }
24     }
25 }
26
27

```

Figure 4.5: Synk Pipeline Stage Implementation In Jenkinsfile

4.3.1 Parameters for Synk Configuration

In the configuration stage shown in figure 4.5 above is for snyk to perform the actual software composition analysis on the codebase, aiming to detect vulnerabilities in its dependencies. The declaration establishes the stage named **"RunSCAAnalysisUsingSnyk"**

delineating its purpose as the execution of Snyk's Software Composition Analysis. The **"withCredentials"** directive facilitates the secure handling of sensitive information by allowing the usage of the Snyk token stored as a Jenkins credential. The **"sh"** step enables the execution of shell commands while the **"snyk:test"** directive initiates the Snyk analysis, which assesses the project's software dependencies for potential security vulnerabilities. Finally **"-fn"** flags appended to the command enhance logging to display all output, even in case of failures.

4.3.2 Snyk Build and Test result

Below is the build and test result of the scan. The pipeline checkout the code and run a SAST scan then a SCA scan on the code

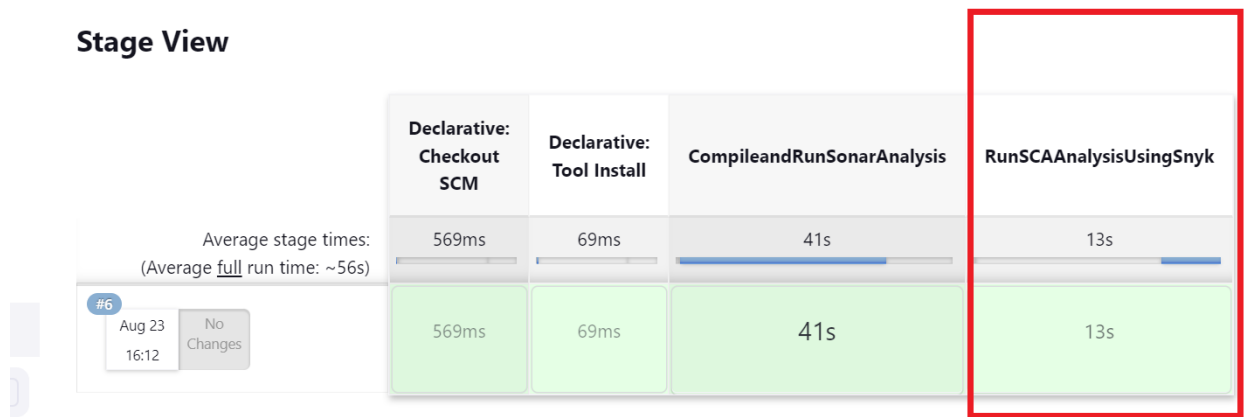


Figure 4.6: Snyk Pipeline Build Result In Jenkins Dashboard

```
[INFO] Snyk CLI Version: 1.1206.0
[INFO]
[INFO] Testing /var/lib/jenkins/workspace/DevSecOps-Pipeline...
[INFO]
[INFO] Tested 35 dependencies for known issues, found 44 issues, 44 vulnerable paths.
[INFO]
[INFO]
[INFO] Issues to fix by upgrading:
[INFO]
[INFO] Upgrade mysql:mysql-connector-java@5.1.25 to mysql:mysql-connector-java@8.0.28 to fix
[INFO] ??? Improper Access Control [Low Severity][https://security.snyk.io/vuln/SNYK-JAVA-MYSQL-31449] in
mysql:mysql-connector-java@5.1.25
[INFO] introduced by mysql:mysql-connector-java@5.1.25
[INFO] ??? Improper Authorization [Medium Severity][https://security.snyk.io/vuln/SNYK-JAVA-MYSQL-2386864] in
mysql:mysql-connector-java@5.1.25
```

Figure 4.7: Snyk Scan Result In Jenkins Console Output

After a successful build, snyk found 35 dependencies for already known issues, 44 other issues and 44 vulnerable paths in the dependencies. Snyk not only identifies vulnerabilities but also provides guidance on how to remediate them. It suggests available patches, alternative package versions, or code changes that can mitigate the security risks. By leveraging Snyk's capabilities to detect vulnerabilities and potential security weaknesses within the project's external dependencies, the pipeline contributes to proactive identification and mitigation of security risks, thereby bolstering the overall robustness and integrity of the software product.

4.4 Building and Containerizing Application

Before performing a DAST scan this is a preliminary stage that is needed to be implemented. In this pipeline, we first build a Docker image to our docker hub named "easy" using the Docker daemon running on the Jenkins machine. Then we push the built image to an Amazon ECR repository using the specified credentials and URL. Docker login to the aws account using docker login with the access and secret access key shown in figure 3.2. The reason we push to ECR is because we will still the containerized application from ECR to be deployed to Kubernetes for orchestration and DAST scanning.

```
3 //building the image
4 stage('Build') {
5     steps {
6         withDockerRegistry(credentialsId: "dockerlogin", url: "") {
7             script{
8                 app = docker.build("easy")
9             }
10        }
11    }
12 }
13
14 //push
15 stage('Push') {
16     steps {
17         script{
18             docker.withRegistry('https://528773513133.dkr.ecr.eu-west-2.amazonaws.com', 'ecr:eu-west-2:aws-credentials') {
19                 app.push("latest")
20             }
21         }
22     }
23 }
```

Figure 4.8: Code Snippet of The Build and Push Stage in the Jenkinsfile

Stage View

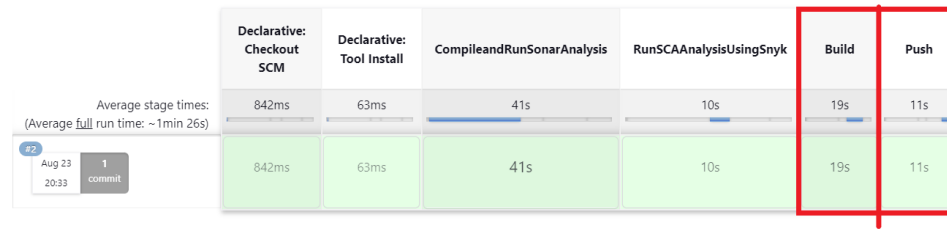


Figure 4.9: Image Showing A successful Jenkins Pipeline With The App Image Built and Push to ECR

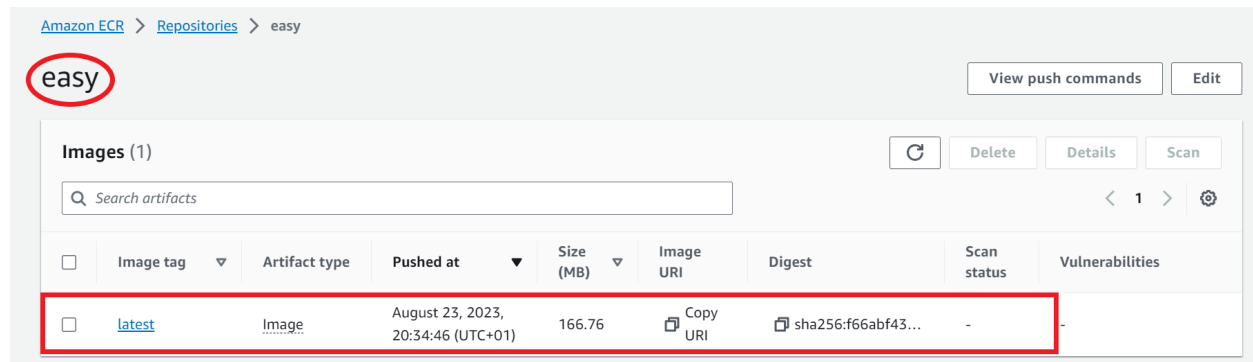


Figure 4.10: Image of The App Dockerized Image Built to Amazon ECR Ready For Deployment to Amazon Kubernetes

4.5 Seamless Deployment of Containerized Application on AWS Kubernetes

After containerizing our Java application and pushing it to ECR by encapsulating all the necessary dependencies within the container image, another preliminary stage needed for the implementation of DAST on our java application is to deploy our application to Kubernetes for a seamless runtime environment making it easy to scale our application horizontally by adding or removing instances of our containers.

```

49
50 stage('Kubernetes Deployment of EasyBuggy Web Application') {
51     steps {
52         withKubeConfig([credentialsId: 'kubelogin']) {
53             sh('kubectl delete all --all -n devsecops')
54             sh('kubectl apply -f deployment.yaml --namespace=devsecops')
55         }
56     }
57 }

```

Figure 4.11: Image of The App Dockerized Image Built to Amazon ECR Ready For Deployment to Amazon Kubernetes

The provided code snippet above is a Jenkins pipeline script written in the Jenkinsfile. This script utilizes the Jenkins Kubernetes plugin to handle the deployment of the application to a Kubernetes cluster. The **withKubeConfig** block is employed to set up the interaction with the Kubernetes cluster. The necessary credentials for accessing Kubernetes are specified using the **credentialsId** parameter, which refers to Jenkins credentials holding the AWS-deployed Kubernetes configuration. The initial step executed involves running a shell command (**kubectl delete all --all -n devsecops**) using the sh step. This command instructs Kubernetes to remove all resources (pods, services, deployments, etc.) within the "devsecops" namespace. On the other hand, the final command applies the **deployment.yaml** file (containing EasyBuggy web application deployment configuration) to the "devsecops" namespace.

Stage View

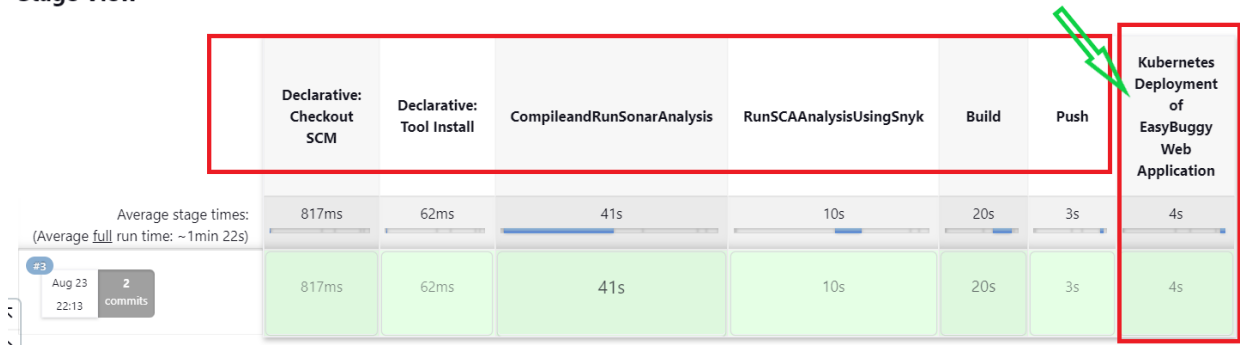


Figure 4.12: Successful Deployment of EasyBuggy Application on Kubernetes

```

aws
Services
Search
[Alt+S]
London
userDevSecOps @ 5287-7351-3

[ec2-user@ip-172-31-38-110 ~]$ kubectl get deployments --namespace=devsecops
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
easybuggy-deployment  1/1     1             1           5m4s
[ec2-user@ip-172-31-38-110 ~]$ kubectl get svc --namespace=devsecops
NAME          TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
easybuggy     LoadBalancer  10.100.43.49  abf0cfbad1db94924a4f9ddf25db684d-539840078.eu-west-2.elb.amazonaws.com  80:30903/TCP    5m11s

```

Figure 4.13: Successful Deployment of EasyBuggy Application on Kubernetes

The figure above shows our application being deployed on Kubernetes with an external IP address (or hostname) of **abf0cfbad1db94924a4f9ddf25db684d-539840078.eu-west-2.elb.amazonaws.com** which is URL where our application can be accessed with the service configured to forward incoming traffic from port 80 to port 30903 on the cluster's nodes.

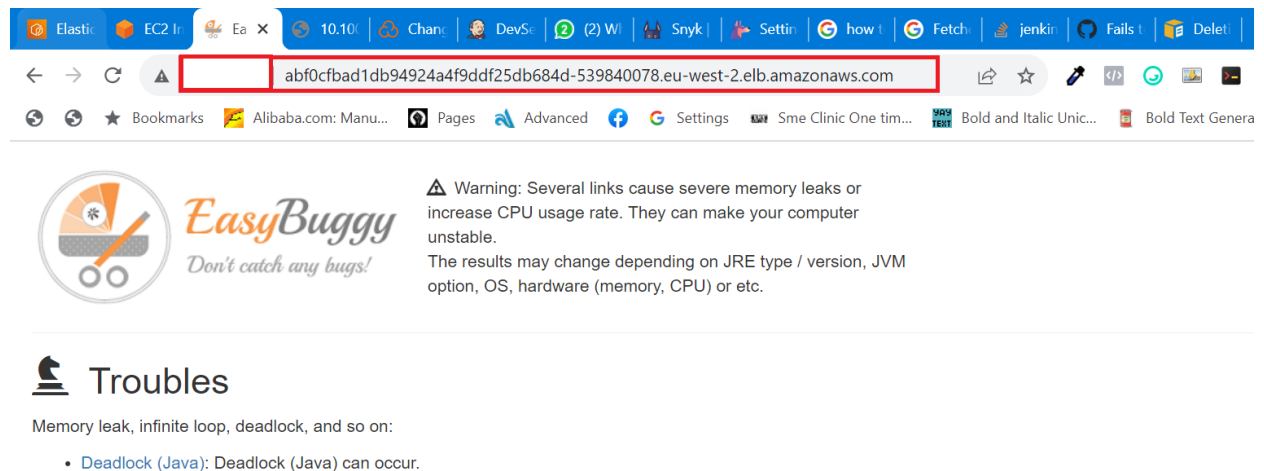


Figure 4.14: EasyBuggy Web Application Up and Running on Kubernetes Cluster

Steps Taken So Far:

- The Initial step entails retrieving the source code from GitLab to Jenkins.
- The subsequent step focuses on establishing the necessary tools essential for the build and analysis procedures. These tools encompass compilers and other dependencies indispensable for a successful software construction.
- The third step encompasses compiling the source code and conducting a Sonar Analysis to evaluate the code's quality by scrutinizing various factors such as bugs, vulnerabilities, and code smells.
- In the fourth step, an examination of the codebase is conducted to identify any known vulnerabilities present in the third-party libraries and components it relies on. The fifth step involves compiling the code, executing tests, and generating executable artifacts that are then stored in our Docker Hub repository.
- In the sixth step, the built artifacts are pushed to Amazon ECR and finally, we deployed the application from Amazon ECR to Kubernetes to ensure a functional runtime environment.

The next task in the DevSecOps pipeline is to perform a DAST scan on our application to simulate real life attack scenarios in our application.

4.6 DAST Implementation

DAST scans serve as a means of simulating the actions of a real attacker on our application, allowing for the identification of vulnerabilities that may go unnoticed in other forms of testing. These scans encompass the entirety of the application, including its front-end and back-end components, APIs, and any integrations. By utilizing OWASP ZAP, we generated a comprehensive reports detailing the severity of identified vulnerabilities and providing recommended steps for remediation.

```
stage('wait_for_testing'){
  steps {
    sh 'pwd; sleep 180; echo "Application Has been deployed on K8S"'
  }
}

stage('RunDASTUsingZAP') {
  steps {
    withKubeConfig([credentialsId: 'kubelogin']) {
      sh('zap.sh -cmd -quickurl http://$(kubectl get services/easybuggy --namespace=devsecops -o json| jq -r ".status.loadBalancer.ingress[] | .hostname") -quickprogress -quickout ${WORKSPACE}/zap_report.html')
      archiveArtifacts artifacts: 'zap_report.html'
    }
  }
}
```

Figure 4.15: EasyBuggy Web Application Up and Running on Kubernetes Cluster

4.6.1 Parameters for OWASP ZAP

From the DAST stage in the Jenkinsfile above, the **wait_for_testing** stage serves as a temporary placeholder for the period of waiting after an application has been deployed on Kubernetes. The code utilizes the sh step to execute shell commands within the pipeline. Specifically, it first prints the current working directory (pwd), then waits for 180 seconds using the sleep command, and finally echoes a message confirming that the application has been successfully deployed on Kubernetes. This stage has been incorporated to ensure that the deployed application is fully operational before commencing security testing. The presence of the **withKubeConfig** block indicates that the pipeline is configured to function with Kubernetes using a designated credentials ID (kubelogin) saved in Jenkins credentials file. Within this stage, the sh step runs the ZAP tool through a command-line

interface (**zap.sh**). It performs a quick scan (-quickurl) on the URL of the deployed application, obtained using kubectl command and jq (a command-line JSON processor) to extract the hostname of the application's load balancer service. The results of this scan are stored in an HTML report named **zap_report.html** to enable later analysis and reporting, the **archiveArtifacts** step archives this ZAP HTML report as an artifact that can be accessed.

4.6.2 OWASP ZAP Build and Test Result

Below is the build and scan result of the DAST scan by OWASP ZAP.

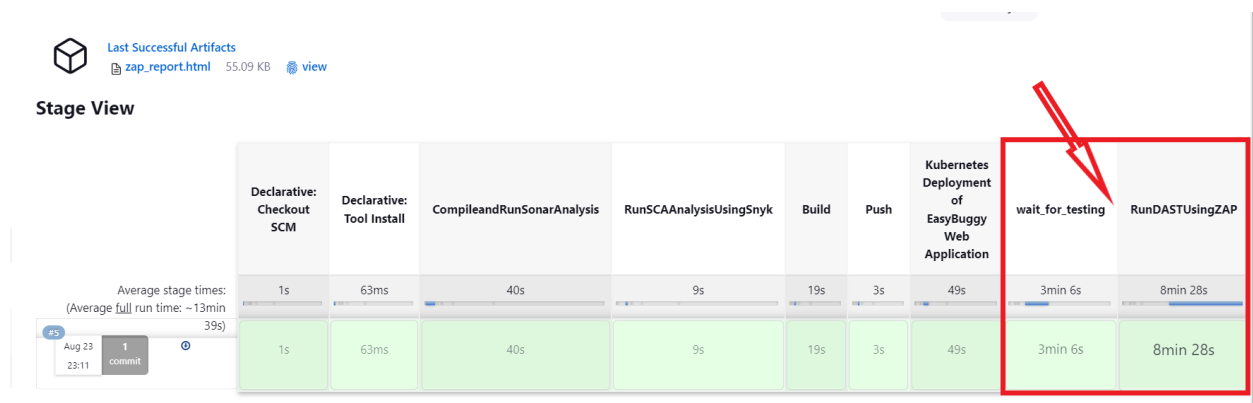


Figure 4.16: OWASP ZAP Pipeline Build Result on Jenkins

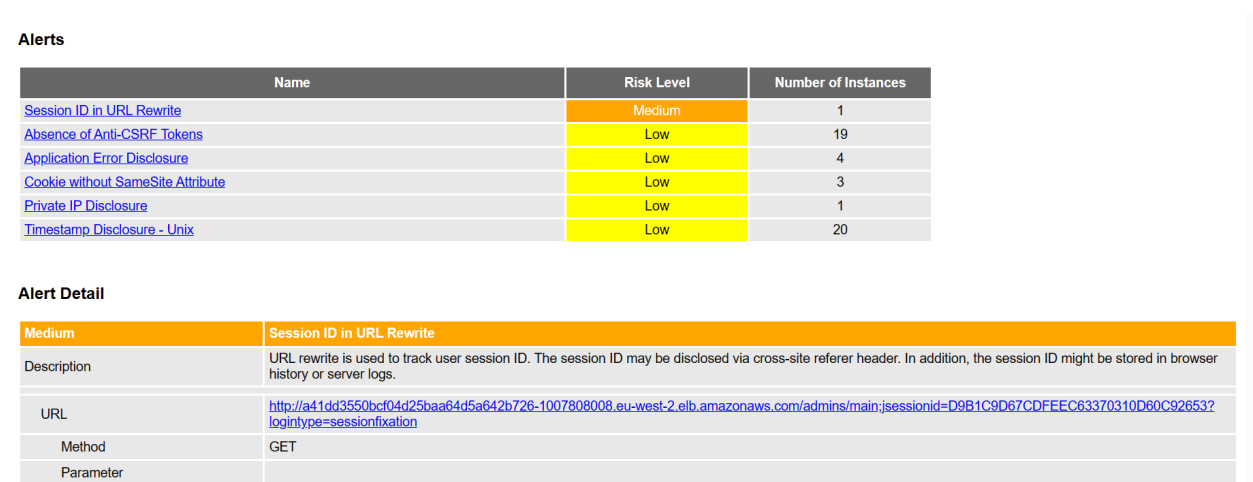


Figure 4.17: OWASP ZAP Scan Test Result

This report presents an analysis of the security weaknesses found in the scanned web

application. It includes an assessment of the risk levels associated with each vulnerability, as well as detailed information on each vulnerability. Additionally, the report offers recommended solutions and references to improve the security of our web application.

4.7 Discussion of Result

4.7.1 Discovered Vulnerabilities

After the security testing phases were finished, a wide range of vulnerabilities were found, which confirms the effectiveness of the integrated security practices. These vulnerabilities were classified according to their severity levels as defined by the Common Vulnerability Scoring System (CVSS). A thorough variety of vulnerabilities were identified, including specific instances of Common Vulnerabilities and Exposures (CVEs) and their corresponding Common Weakness Enumeration (CWE) identifiers. Although SonarCloud found some vulnerabilities with no security associated CWEs.

High Vulnerabilities: SonarCloud and Snyk unveiled high-severity vulnerabilities. These include CWE-89 (SQL Injection) and CWE-200 (Security Misconfiguration) respectively. These vulnerabilities have the potential to compromise data integrity and confidentiality, exposing applications to malicious attacks. SQL Injection occurs when an attacker manipulates an application's input fields to inject malicious SQL code into the application's database query. This can lead to unauthorized access to the database, data leakage, data manipulation, and even potential damage to the application or database (OWASP n.d.).

Medium Vulnerabilities: SonarCloud and Zap in Unison detected a number of medium-severity vulnerabilities, including CWE-310 (related to cryptographic issues) and CWE-200 (involving the exposure of sensitive information to unauthorized individuals). CWE-200 specifically highlights the utilization of URL rewriting to monitor user session IDs. This practice can potentially lead to the disclosure of session IDs through cross-site referrer headers. Furthermore, there is a risk that session IDs could be stored in browser histories or server logs.

Low Vulnerabilities: SonarCloud, ZAP, and Snyk collectively identified low-severity vulnerabilities (CWE-352, CWE-1275) predominantly associated with cross-site request

forgery (CSRF). CSRF refers to an attack where the victim unknowingly or unintentionally sends an HTTP request to a specific destination, enabling unauthorized actions in the guise of the victim (OWASP n.d.).

4.7.2 Comparison of The Security Testing Techniques

The three security testing techniques, namely Static Application Security Testing (SAST), Software Composition Analysis (SCA), and Dynamic Application Security Testing (DAST), provide unique perspectives on an application's vulnerabilities. The efficacy of these techniques in detecting various vulnerabilities and their influence on the overall security stance can differ depending on the specific vulnerabilities they focus on and their integration within the software development lifecycle.

The inclusion of SAST in the pipeline proved to be an effective means of identifying a significant amount of vulnerabilities within the source code. Nevertheless, there were instances where **false positives** were generated, mistakenly flagging certain code segments as vulnerabilities. This emphasizes the necessity of adjusting the SAST tool to align with the unique coding practices employed by the development team. Furthermore, although SAST excels in its ability to detect established vulnerabilities, it encounters difficulties in identifying logical vulnerabilities that necessitate an understanding of the application's behavioral context. The time it takes for all the scan result above in figure 4.2, 4.6 and 4.16 shows that it takes an average of 40s for SAST scan to run based on the web application code size.

Snyk's integration for software composition analysis (SCA) effectively detected vulnerabilities in third-party dependencies. Nevertheless, it revealed a significant concern: a substantial number of these vulnerabilities were a result of using outdated versions of libraries. This highlights the significance of continuous dependency management. Furthermore, SCA tools can gain advantages from improved visibility into the software supply chain, potentially offering valuable information regarding the security of upstream dependencies. The time it takes for all the scan result above in figure 4.2, 4.6 and 4.16 shows that it takes an average of 9s for SCA scan to run based on the web application because it only just checks of third party libraries and dependencies used in the application code.

The utilization of OWASP Zap for DAST presented several difficulties in the integration process. The completion of DAST typically takes longer due to the necessity of runtime interaction with the application, as depicted in figure 4.16. This can cause delays in the

pipeline, particularly for intricate applications. Although DAST effectively detects vulnerabilities that occur during runtime, it occasionally overlooks certain problems that cannot be easily identified through automated methods. To enhance the efficiency of DAST, a strategic approach to selecting scan targets and conducting scans during off-peak periods could be taken into account.

As mentioned earlier, and from the build time of each testing techniques it shows that DAST take more time than SAST to be deployed this as a result of simulating real world attack on the running application. While some organizations want to deploy quickly running full DAST scan during deploy everytime might slow down their production level. Adjustments are necessary in cases where conducting a full DAST scan within a reasonable time-frame is not feasible. To address this issue, there are multiple avenues to consider. Firstly, optimizing the intensity of the DAST test can be achieved through strategies such as limiting the number of requests generated by the tool or focusing assessments on specific components of the application. Furthermore, customizing different deployment environments with varying levels of testing intensity can be highly beneficial in reducing the duration of build jobs. For example, build processes aimed at staging environments may not require the same level of thorough examination as those intended for production workloads. This customized approach ensures efficient utilization of resources and aligns testing efforts with deployment requirements.

The frequency of scans can be determined based on the pace of development. SAST and SCA scans can be incorporated more frequently as part of Continuous Integration/Continuous Deployment (CI/CD) pipelines, whereas DAST scans can be scheduled periodically or during significant updates.

Tables 4.1 below shows the advantages of each security testing techniques.

Technique	Advantages	Disadvantages
SAST	<ul style="list-style-type: none"> • Early detection of vulnerabilities at the source code level. • Identification of complex vulnerabilities, including design flaws. • Integration into the development process for continuous feedback. • Ability to find security issues before compilation. 	<ul style="list-style-type: none"> • Potential for false positives and false negatives. • Dependent on accurate and up-to-date code. • May miss runtime-specific vulnerabilities.
SCA	<ul style="list-style-type: none"> • Identification of vulnerabilities in third-party components. • Assessment of licensing and compliance issues. • Visibility into the software supply chain. • Reduction of risks from external dependencies. 	<ul style="list-style-type: none"> • May not identify vulnerabilities in proprietary code. • Relies on accurate vulnerability databases. • Challenges in tracking and updating third-party components. • Does not cover runtime behavior.
DAST	<ul style="list-style-type: none"> • Real-time identification of vulnerabilities during application runtime. • Simulation of real-world attack scenarios. • Testing of application behavior in a live environment. • Identification of runtime-specific vulnerabilities. 	<ul style="list-style-type: none"> • Can be time-consuming for large codebases. • May produce false positives and negatives due to dynamic nature. • Does not uncover vulnerabilities in source code. • Cannot identify design or architectural vulnerabilities.

Table 4.1: Advantages and Disadvantages of Each Security Testing Techniques

4.8 Conclusion

The findings discussed in this chapter illustrate a pragmatic model that organizations can use to incorporate security measures into the development process. By placing importance on early identification of vulnerabilities, careful monitoring of third-party dependencies, simulation of real-world attacks, and practical integration of security practices, organizations can create applications that are more secure without compromising development speed. The advantages and disadvantages associated with each testing technique,

as well as the overall integration of DevSecOps practices, provide a balanced view of the challenges and benefits that organizations can expect. This practical approach helps bridge the gap between theory and practice, empowering organizations to actively improve software security within the ever-changing landscape of software development.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Integrating SAST, SCA, and DAST in the DevSecOps pipeline showed how effective it is at finding vulnerabilities in the whole software development process. By using these methods together, the pipeline covered everything, including code weaknesses, problems with third-party components, and vulnerabilities at runtime. The collaboration between development, security, and operations teams in the DevSecOps pipeline helped detect and fix vulnerabilities early on. By incorporating SAST and SCA stages into the development and CI/CD process, the introduction of vulnerabilities was greatly reduced.

Additionally, DAST played a vital role in identifying runtime vulnerabilities, even in later stages. Comparing SAST, SCA, and DAST techniques gives us an understanding of their individual strengths and limitations. This knowledge can help organizations choose the right testing techniques according to their requirements. The research provides a detailed methodology and implementation that organizations can use as a blueprint for implementing a DevSecOps pipeline. Following this guidance can help organizations adopt effective security practices more quickly.

5.2 Limitations and Future Work

5.2.1 Limitations of the Study

While this research provides valuable insights into the integration of security testing within the DevSecOps pipeline, There are certain limitations that influenced the outcomes and conclusions of the study.

- One major limitation of the study is its lack of specificity. The focus was only on deploying a web application on AWS infrastructure using a predefined toolkit. This means that the findings can only be applied to scenarios that are similar to this specific setup. Therefore, applying these findings to different cloud environments, toolchains, or hybrid infrastructures requires careful consideration. The narrow scope of the study raises questions about how adaptable and scalable the suggested DevSecOps pipeline is.
- The choice of tools used for SAST, SCA, and DAST affects the effectiveness of the DevSecOps pipeline. The tools we chose for this study were determined by specific criteria, but there are many other tools to choose from. Using different tools could produce different outcomes, which would impact the overall effectiveness of the pipeline.

5.2.2 Future Work

In order to expand the integration of security, future research can explore a wider range of testing techniques that can enhance the DevSecOps pipeline are as follows

- An interesting area for further investigation is incorporating Fuzz Testing into the pipeline. Fuzz testing involves subjecting an application to unexpected or random inputs to uncover vulnerabilities that may not be found through traditional testing methods.
- It is important to also consider Interactive Application Security Testing (IAST) as a valuable addition to the DevSecOps pipeline, which was mentioned in chapter 2 of

this research. IAST combines elements of SAST and DAST by examining application behavior while it is running, like DAST does, and also identifying vulnerabilities in specific lines of code, similar to SAST. However, IAST is not implemented in this dissertation.

- Another area worth exploring is the integration of Runtime Application Self-Protection (RASP) mechanisms. Unlike traditional testing methods that are used before deployment, RASP operates within the deployed application itself. It continuously monitors application behavior in real-time and can identify and mitigate attacks as they happen.

REFERENCES

Accenture (n.d.) *What is Cloud Computing & Why is it Important?*.

<https://www.accenture.com/gb-en/cloud/insights/cloud-computing-index> Accessed 13 July 2023.

Adnan, S., Moin, K. and Imran, J. (2023) DevOps with Agile: Best Practices to Improve Software Quality. *International Conference on Biological Research and Applied Science*. Jinnah University for Women, Karachi, Pakistan.

Akbar, M.A., Smolander, K., Mahmood, S. and Alsanad, A. (2022) Toward successful DevSecOps in software development organizations: A decision-making framework. *Information and Software Technology* 147, 106894.

Alawneh, M. and Abbadi, I.M. (2022) Expanding DevSecOps Practices and Clarifying the Concepts within Kubernetes Ecosystem. *2022 Ninth International Conference on Software Defined Systems (SDS)* IEEE.

Albailhaqi, M.F., Wilda, A.N. and Sugiantoro, B. (2020) Deploying an Application to Cloud Platform Using Continuous Integration and Continuous Delivery. *Proceeding International Conference on Science and Engineering* 3, 279–282.

Anand, P., Ryoo, J., Kim, H. and Kim, E. (2016) Threat Assessment in the Cloud Environment. *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication* New York, NY, USA: ACM.

Anjaria, D. and Kulkarni, M. (2022) Effective DevSecOps Implementation: A Systematic Literature Review. *CARDIOMETRY* (24), 410–417.

AWS (n.d.) *Elasticity - AWS Well-Architected Framework*. <https://rebrand.ly/awselas> Accessed 18 July 2023.

AWS (n.d.) *Secure and resizable cloud compute – Amazon EC2 – Amazon Web Services*. <https://aws.amazon.com/ec2/> Accessed 18 July 2023

Bolscher, R. and Daneva, M. (2019) Designing Software Architecture to Support Continuous Delivery and DevOps: A Systematic Literature Review. *Proceedings of the 14th*

International Conference on Software Technologies SCITEPRESS - Science and Technology Publications.

Bush, C. (2021) *Continuous Security: Threat Modeling in DevSecOps*.
<https://bishopfox.com/blog/threat-modeling-in-devsecops> Accessed 15 July 2023.

Cameron, C. (2023) *SonarCloud or SonarQube? - Guidance on Choosing One for Your Team*. Sonar 15 May.

Candel, J.M.O. (2022) *Implementing DevSecOps with Docker and Kubernetes: An Experiential Guide to Operate in the DevOps Environment for Securing and Monitoring Container Applications (English Edition)*. BPB Publications.

Carter, K. (2017) Francois Raynaud on DevSecOps. *IEEE Software* 34 (5), 93–96.

Carturan, S.B.O.G. and Goya, D.H. (2019) A systems-of-systems security framework for requirements definition in cloud environment. *Proceedings of the 13th European Conference on Software Architecture - Volume 2* New York, NY, USA: ACM.

Chandramouli, R. (2022) *Implementation of DevSecOps for a microservices-based application with service mesh*. Gaithersburg, MD: National Institute of Standards and Technology (U.S.).

Cho, G., Choi, J., Kim, H., Hyun, S. and Ryoo, J. (2019) *Threat Modeling and Analysis of Voice Assistant Applications*. https://link.springer.com/chapter/10.1007/978-3-030-17982-3_16 Accessed 13 July 2023.

Cois, C.A., Yankel, J. and Connell, A. (2014) Modern DevOps: Optimizing software development through effective system interactions. *2014 IEEE International Professional Communication Conference (IPCC)* IEEE.

Creane, B. and Gupta, A. (2021) *Kubernetes Security and Observability*. “O’Reilly Media, Inc.”

Curphey, M. (2019) *Fail Fast: How Shifting Security Left Speeds Development*.
<https://devops.com/fail-fast-how-shifting-security-left-speeds-development/> Accessed 10 July 2023

Daniel, W. (2020) *Zed Attack Proxy (ZAP)*.

Das, B.K. and Chu, V. (2023) *Security As Code: DevSecOps Patterns with AWS*. O’Reilly Media.

- Dhawan, V. and Sabetto, R. (2019) *DevSecOps-Security and Test Automation Briefing*. <https://www.mitre.org/news-insights/publication/devsecops-security-and-test-automation-briefing> Accessed 4 August 2023.
- Diel, E., Marczak, S. and Cruzes, D.S. (2016) Communication Challenges and Strategies in Distributed DevOps. *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)* IEEE.
- Ebert, C., Gallardo, G., Hernantes, J. and Serrano, N. (2016) DevOps. *IEEE Software* 33 (3), 94–100.
- EC-Council (2023) *What is Cyber Threat Modeling*. <https://www.eccouncil.org/threat-modeling/> Accessed 15 July 2023.
- Fitzgerald, B. and Stol, K.-J. (2017) Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software* 123, 176–189.
- Garousi, V. and Mäntylä, M.V. (2016) When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology* 76, 92–117.
- Gartner (2016) *DevSecOps: How to Seamlessly Integrate Security Into DevOps*.
- GitHub (n.d.) *GitHub features: the right tools for the job*. <https://github.com/features> Accessed 19 August 2023.
- Guo, T. (2021) DevSecOps Introduction for beginners: Security in the SDLC - *GitGuardian Blog*. *GitGuardian Blog - Automated Secrets Detection* 10 May.
- Gupta, A. (2022) An Integrated Framework for DevSecOps Adoption. *International Journal of Computer Trends and Technology* 70 (6), 19–23.
- Hashicorp (n.d.) What is Infrastructure as Code with Terraform? *What is Infrastructure as Code with Terraform? — Terraform — HashiCorp Developer*.
- Hsu, T. (2018) *Hands-On Security in DevOps: Ensure continuous security, deployment, and delivery with DevSecOps*. Packt Publishing Ltd.
- Humble, J. and Molesky, J. (2011) Why Enterprises Must Adopt DevOps to Enable Continuous Delivery. *The Journal of Information Technology Management* 6–12.
- Hussain, S., Kamal, A., Ahmad, S., Rasool, G. and Iqbal, S. (2014) *Threat Modelling Methodologies: a Survey*.

- Ibrahim, A., Yousef, A.H. and Medhat, W. (2022) DevSecOps: A Security Model for Infrastructure as Code Over the Cloud. *2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)* IEEE.
- Ifrah, S. (2019) *Scaling the AWS EKS, ECS, and ECR Containerized Environments*. Berkeley, CA: Apress 255–288.
- Jangla, K. (2018) Docker Images. *Accelerating Development Velocity Using Docker*, Berkeley, CA: Apress. 55–76.
- Jenkins (n.d.) Jenkins User Documentation. *Jenkins User Documentation*.
- Jensen, M., Kapila, S. and Gruschka, N. (2019) Towards Aligning GDPR Compliance with Software Development: A Research Agenda. *Proceedings of the 5th International Conference on Information Systems Security and Privacy* SCITEPRESS - Science and Technology Publications.
- Khan, M.O. (2020) Fast Delivery, Continuously Build, Testing and Deployment with DevOps Pipeline Techniques on Cloud. *Indian Journal of Science and Technology* 13 (5), 552–575.
- Kim, Gene, Behr, K. and Spafford, G. (2018) *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*. Center for Open Science.
- Kraemer, S., Carayon, P. and Clem, J. (2009) Human and organizational factors in computer and information security: Pathways to vulnerabilities. *Computers & Security* 28 (7), 509–520.
- k-tamura (n.d.) *GitHub - k-tamura/easybuggy: Too buggy web application*. <https://github.com/k-tamura/easybuggy> Accessed 20 August 2023.
- Kumar, R. and Goyal, R. (2020) Modeling continuous security: A conceptual model for automated DevSecOps using open-source software over cloud (ADOC). *Computers & Security* 97, 101967.
- Leite, L., Rocha, C., Kon, F., Milojicic, D. and Meirelles, P. (2019) A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys* 52 (6), 1–35.
- Logz (n.d.) *Logz.io*. <https://logz.io/solutions/cloud-monitoring-aws/> Accessed 12 August 2023.
- MacDonald, N. and Head, I. (2016) *DevSecOps: How to Seamlessly Integrate Security Into DevOps*.

- Mansfield-Devine, S. (2018) DevOps: finding room for security. *Network Security* 2018 (7), 15–20.
- Mell, P.M. and Grance, T. (2011) *The NIST definition of cloud computing*. Gaithersburg, MD: National Institute of Standards and Technology.
- Microsoft (2022) *Security in DevOps (DevSecOps) - Azure DevOps*. <https://rebrand.ly/microsoftdevops> Accessed 8 July 2023.
- Microsoft (2023) *What is DevOps? - Azure DevOps*. <https://learn.microsoft.com/en-us/devops/what-is-devops> Accessed 8 July 2023.
- Mishra, P. (2023) Getting Started with AWS. *Cloud Computing with AWS* Berkeley, CA: Apress. 35–72.
- Mohan, V. and Othmane, L.B. (2016) SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps. *2016 11th International Conference on Availability, Reliability and Security (ARES)* IEEE.
- Mohanan, Remya (2022) *What Is DevOps Lifecycle? Definition, Key Components, and Management Best Practices*. <https://www.spiceworks.com/tech/devops/articles/what-is-devops-lifecycle/> Accessed 8 July 2023.
- Myrbakken, H. and Colomo-Palacios, R. (2017) DevSecOps: A Multivocal Literature Review. *Communications in Computer and Information Science* Cham: Springer International Publishing. 17–29.
- Omer, R. (2014) *The Costs of Cloud Migration*. <https://rebrand.ly/iee> Accessed 19 July 2023
- OWASP (n.d.) *OWASP Top Ten*. <https://owasp.org/www-project-top-ten/> Accessed 13 July 2023.
- OWASP (n.d.) *SQL Injection*. <https://owasp.org/www-community/attacks/SQLInjection> Accessed 29 August 2023.
- Parashar, A., Dwivedi, A., Kumar, A. and Ahmad Khan, A. (2020) DevSecOps: A Case Study on A Sample Implementation of DevSecOps. *International Journal of Engineering Applied Sciences and Technology* 5 (2), 156–158.
- Peltier, T.R. (2005) *Information Security Risk Analysis, Second Edition*. CRC Press.
- Puppet (2019) *Download the 2021 State of DevOps Report*. <https://www.puppet.com/resources/state-of-devops-report> Accessed 4 August 2023.

- Rajapakse, R.N., Zahedi, M., Babar, M.A. and Shen, H. (2022) Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology* 141, 106700.
- Rangnau, T., Buijtenen, R. v., Fransen, F. and Turkmen, F. (2020) Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)* IEEE.
- Saini, H., Upadhyaya, A. and Khandelwal, M.K. (2019) Benefits of Cloud Computing for Business Enterprises: A Review. *SSRN Electronic Journal* .
- Salecha, R. (2022) Introduction to Terraform. *Practical GitOps* Berkeley, CA: Apress. 67–122.
- Sarycheva, Y. (2019) *What is Waterfall Model in SDLC? Phases, pros an cons.* <https://xbsoftware.com/blog/software-development-life-cycle-waterfall-model/> Accessed 8 July 2023.
- Shostack, A. (2014) *Threat Modeling: Designing for Security*. John Wiley & Sons.
- Snyk (2022) Guide to Software Composition Analysis (SCA). *Snyk* 27 April.
- Snyk (2021) SAST vs. SCA testing: What's the difference? Can they be combined? *Snyk* 10 October.
- Sonar (n.d.) *Clean code in your cloud workflow with SonarCloud.* <https://www.sonarsource.com/products/sonarcloud/> Accessed 12 August 2023.
- Souppaya, M. (2017) Draft NIST SP 800-190, Application Container Security Guide. *NIST Special Publication (SP) 800-190*.
- Splunk (n.d.) *A Beginner's Guide to Kubernetes Monitoring*.
- Takabi, H., Joshi, J.B.D. and Ahn, G.-J. (2010) Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security & Privacy Magazine* 8 (6), 24–31.
- Tarandach, I. and Coles, M.J. (2020) *Threat Modeling*. O'Reilly Media.
- Tudela, F.M., Higuera, J.-R.B., Higuera, J.B., Montalvo, J.-A.S. and Argyros, M.I. (2020) 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), .

- UcedaVelez, T. and Morana, M.M. (2015) *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. John Wiley & Sons.
- Ur Rahman, A.A. and Williams, L. (2016) Software security in DevOps. *Proceedings of the International Workshop on Continuous Software Evolution and Delivery* New York, NY, USA: ACM.
- Uzayr, S. (2022) *Introduction to Git and GitHub*. Boca Raton: CRC Press, 1–52.
- Valdés-Rodríguez, Y., Hochstetter-Diez, J., Díaz-Arancibia, J. and Cadena-Martínez, R. (2023) Towards the Integration of Security Practices in Agile Software Development: A Systematic Mapping Review. *Applied Sciences* 13 (7), 4578.
- Williams, L. (2018) Continuously integrating security. *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment* New York, NY, USA: ACM.
- van Wyk, K.R. and McGraw, G. (2005) Bridging the Gap between Software Development and Information Security. *IEEE Security and Privacy Magazine* 3 (5), 75–79.
- Yasar, H. (2018) Experiment: Sizing Exposed Credentials in GitHub Public Repositories for CI/CD. *2018 IEEE Cybersecurity Development (SecDev)* IEEE.
- Yasar, H. and Kontostathis, K. (2016) Where to Integrate Security Practices on DevOps Platform. *International Journal of Secure Software Engineering* 7 (4), 39–50.
- Yasar, H. (2020) *Overcoming DevSecOps Challenges: A Practical Guide for All Stakeholders*. <https://apps.dtic.mil/sti/pdfs/AD1110359.pdf>.
- Zap (n.d.) *ZAP – Getting Started*. <https://www.zaproxy.org/getting-started/> Accessed 11 August 2023.
- Zeeshan, A.A. (2020) *DevSecOps for .NET Core*. Berkeley, CA: Apress.

Appendix A

AN APPENDIX

A.1 Deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: easybuggy-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: easybuggy
  template:
    metadata:
      labels:
        app: easybuggy
    spec:
      containers:
        - name: easybuggy
          image: 528773513133.dkr.ecr.eu-west-2.amazonaws.com/easy
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
# service type loadbalancer
```

Figure A.1: Configuration file for eassybuggy containerized applications on Kubernetes

```

24  apiVersion: v1
25  kind: Service
26  metadata:
27    labels:
28      app: easybuggy
29      k8s-app: easybuggy
30    name: easybuggy
31  spec:
32    ports:
33      - name: http
34        port: 80
35        protocol: TCP
36        targetPort: 8080
37    type: LoadBalancer
38    selector:
39      app: easybuggy

```

Figure A.2: Configuration file for easybuggy containerized applications on Kubernetes - cont'd

A.2 Jenkinsfile

```

Jenkinsfile-2
1  pipeline {
2    agent any
3    tools {
4      maven 'Maven_3_5_2'
5    }
6    environment {
7      SONAR_TOKEN = credentials('SONAR_TOKEN')
8    }
9
10   //SONAR ANALYSIS FOR SAST SCAN
11   stages{
12     stage('CompileandRunSonarAnalysis') {
13       steps {
14         sh 'mvn clean verify sonar:sonar -Dsonar.projectKey=devsecopsjenkins -Dsonar.organization=devsecopsjenkins -Dsonar.host.url=https://sonarcloud.io -Dsonar.token=${SONAR_TOKEN}'
15       }
16     }
17
18     //SYNK ANALYSIS FOR SCS SCAN
19     stage('RunSCAAAnalysisUsingSnyk') {
20       steps {
21         withCredentials([string(credentialsId: 'SNYK_TOKEN', variable: 'SNYK_TOKEN')]) {
22           sh 'mvn snyk:test -fn'
23         }
24       }
25     }
26   }

```

Figure A.3: Jenkins configuration file for security build stages

```

27
28 //building the image
29 stage('Build') {
30     steps {
31         withDockerRegistry([credentialsId: "dockerlogin", url: ""]) {
32             script{
33                 app = docker.build("easy")
34             }
35         }
36     }
37 }
38
39 //push
40 stage('Push') {
41     steps {
42         script{
43             docker.withRegistry('https://528773513133.dkr.ecr.eu-west-2.amazonaws.com', 'ecr:eu-west-2:aws-credentials') {
44                 app.push("latest")
45             }
46         }
47     }
48 }
49

```

Figure A.4: Jenkins configuration file for security build stages-cont'd

```

50 stage('Kubernetes Deployment of EasyBuggy Web Application') {
51     steps {
52         withKubeConfig([credentialsId: 'kubelogin']) {
53             sh('kubectl delete all --all -n devsecops')
54             sh('kubectl apply -f deployment.yaml --namespace=devsecops')
55         }
56     }
57 }
58
59 stage('wait_for_testing'){
60     steps {
61         sh 'pwd; sleep 180; echo "Application Has been deployed on K8S"'
62     }
63 }
64
65 stage('RunDASTUsingZAP') {
66     steps {
67         withKubeConfig([credentialsId: 'kubelogin']) {
68             sh('zap.sh -cmd -quickurl http://$(kubectl get services/easybuggy --namespace=devsecops -o json | jq -r ".status.loadBalancer.ingress[] | .hostname") -quickprogress -quickout ${WORKSPACE}/zap_report.html')
69             archiveArtifacts artifacts: 'zap_report.html'
70         }
71     }
72 }
73
74 }
75

```

Figure A.5: Jenkins configuration file for security build stages-cont'd

A.3 Dockerfile

```
Dockerfile > FROM
1 FROM maven:3.8-jdk-8 as builder
2 COPY . /usr/src/easybuggy/
3 WORKDIR /usr/src/easybuggy/
4 RUN mvn -B package
5
6 FROM openjdk:8-slim
7 COPY --from=builder /usr/src/easybuggy/target/easybuggy.jar /
8 CMD ["java", "-XX:MaxMetaspaceSize=128m", "-Xloggc:logs/gc_%p_%t.log", "-Xmx256m", "-XX:MaxDirectMemorySize=90m",
"-XX:+UseSerialGC", "-XX:+PrintHeapAtGC", "-XX:+PrintGCDetails", "-XX:+PrintGCDateStamps", "-XX:
+UseGCLogFileRotation", "-XX:NumberOfGCLogFiles=5", "-XX:GCLogFileSize=10M", "-XX:GCTimeLimit=15",
"-XX:GCHeapFreeLimit=50", "-XX:+HeapDumpOnOutOfMemoryError", "-XX:HeapDumpPath=logs/", "-XX:ErrorFile=logs/
hs_err_pid%p.log", "-agentlib:jdwp=transport=dt_socket,server=y,address=9009,suspend=n", "-Dderby.stream.error.
file=logs/derby.log", "-Dderby.infolog.append=true", "-Dderby.language.logStatementText=true", "-Dderby.locks.
deadlockTrace=true", "-Dderby.locks.monitor=true", "-Dderby.storage.rowLocking=true", "-Dcom.sun.management.
jmxremote", "-Dcom.sun.management.jmxremote.port=7900", "-Dcom.sun.management.jmxremote.ssl=false", "-Dcom.sun.
management.jmxremote.authenticate=false", "-ea", "-jar", "easybuggy.jar"]
9
```

Figure A.6: Config file used to build the docker container image for the application

A.4 OWASP ZAP Result

Low	Absence of Anti-CSRF Tokens
Description	<p>No Anti-CSRF tokens were found in a HTML submission form.</p> <p>A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf.</p> <p>CSRF attacks are effective in a number of situations, including:</p> <ul style="list-style-type: none">* The victim has an active session on the target site.* The victim is authenticated via HTTP auth on the target site.* The victim is on the same local network as the target site. <p>CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.</p>

Figure A.7: Vulnerability result and its associated CWE ID

	<p>Do not use the GET method for any request that triggers a state change.</p> <p>Phase: Implementation</p> <p>Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.</p>
Reference	http://projects.webappsec.org/Cross-Site-Request-Forgery http://cwe.mitre.org/data/definitions/352.html
CWE Id	352
WASC Id	9
Plugin Id	10202

Figure A.8: Vulnerability result and its associated CWE ID - cont'd

A.5 Terraform Declarative File

```
main.tf > resource "aws_security_group" "jenkins_sg" > name
1  terraform {
2    required_version = ">= 0.12"
3  }
4
5  #the cloud provider I want to use
6  provider "aws" {
7    region = var.aws_region
8  }
9
10 #the aws region which has been define in the varribae file "dev-west-2.tfvars"
11 variable "aws_region" {
12   type = string
13 }
14
15 #the aws vpc which has been define in the varribae file "dev-west-2.tfvars"
16 variable "vpc_id" {
17   type = string
18 }
19
20 #the aws region which has been define in the varribae file "dev-west-2.tfvars"
21 variable "key_name" {
22   type = string
23 }
24
25 # this will create security group and open port 8081 for jenkins
26 resource "aws_security_group" "jenkins_sg" {
27   name        = "jenkins_sg"
28   description = "Allow Jenkins Traffic"
29   vpc_id      = var.vpc_id
30 }
```

Figure A.9: Terraform script used for deploying AWS resources

```
30
31 ingress {
32   description = "Allow from Personal CIDR block"
33   from_port   = 8081
34   to_port     = 8081
35   protocol    = "tcp"
36   cidr_blocks = ["0.0.0.0/0"]
37 }
38
39 ingress {
40   description = "Allow SSH from Personal CIDR block"
41   from_port   = 22
42   to_port     = 22
43   protocol    = "tcp"
44   cidr_blocks = ["0.0.0.0/0"]
45 }
46
47 egress {
48   from_port   = 0
49   to_port     = 0
50   protocol    = "-1"
51   cidr_blocks = ["0.0.0.0/0"]
52   ipv6_cidr_blocks = [":::/0"]
53 }
54
55 tags = {
56   Name = "Jenkins SG"
57 }
58 }
```

Figure A.10: Terraform script used for deploying AWS resources - cont'd

```

60 data "aws_ami" "amazon_linux" {
61     most_recent = true
62
63     filter {
64         name = "name"
65         values = ["amzn2-ami-hvm-2.0*"]
66     }
67
68     filter {
69         name = "virtualization-type"
70         values = ["hvm"]
71     }
72
73     filter {
74         name = "root-device-type"
75         values = ["ebs"]
76     }
77
78     owners = ["amazon"] # Canonical
79 }
80
81 resource "aws_iam_role" "test_role" {
82     name = "test_role"
83
84     assume_role_policy = <<EOF
85     {
86         "Version": "2012-10-17"
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Figure A.11: Terraform script used for deploying AWS resources - cont'd

```

81 resource "aws_iam_role" "test_role" {
82     name = "test_role"
83
84     assume_role_policy = <<EOF
85     {
86         "Version": "2012-10-17",
87         "Statement": [
88             {
89                 "Action": "sts:AssumeRole",
90                 "Principal": {
91                     "Service": "ec2.amazonaws.com"
92                 },
93                 "Effect": "Allow",
94                 "Sid": ""
95             }
96         ]
97     }
98     EOF
99 }
100
101 resource "aws_iam_instance_profile" "test_profile" {
102     name = "test_profile"
103     role = "${aws_iam_role.test_role.name}"
104 }
105
106 resource "aws_iam_role_policy" "test_policy" {
107     name = "test_policy"
108     role = "${aws_iam_role.test_role.id}"
109 }
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Figure A.12: Terraform script used for deploying AWS resources - cont'd

```

107     name = "${aws_iam_policy.test_policy.name}"
108     role = "${aws_iam_role.test_role.id}"
109
110     policy = <<EOF
111     {
112         "Version": "2012-10-17",
113         "Statement": [
114             {
115                 "Effect": "Allow",
116                 "Action": "*",
117                 "Resource": "*"
118             }
119         ]
120     }
121     EOF
122 }
123
124 resource "aws_instance" "web" {
125     ami           = data.aws_ami.amazon_linux.id
126     instance_type = "t2.xlarge"
127     key_name      = var.key_name
128     iam_instance_profile = "${aws_iam_instance_profile.test_profile.name}"
129     security_groups = [aws_security_group.jenkins.sg.name]
130     user_data       = "${file("install_jenkins_zap.sh")}"
131     tags = {
132         Name = "Jenkins EC2"
133     }
134 }

```

Figure A.13: Terraform script used for deploying AWS resources - cont'd

A.6 Shell Script

```

$ install_jenkins_zap.sh
1  #!/bin/bash
2
3  # Install Updated packages on linux machine
4  sudo yum update -y
5  sudo wget -O /etc/yum.repos.d/jenkins.repo \
6  | https://pkg.jenkins.io/redhat-stable/jenkins.repo
7  sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key
8  sudo yum upgrade
9
10
11 #sudo yum install jenkins java-1.8.0-openjdk-devel -y
12 sudo amazon-linux-extras install java-openjdk11
13 sudo yum install git -y
14 sudo wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apac
15 sudo sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo
16 sudo yum install -y apache-maven
17 sudo yum install jenkins -y
18 sudo sed -i -e 's/Environment="JENKINS_PORT=[0-9]\{+\}/Environment="JENKINS_PORT=8081"/' /usr/lib/systemd/system/jenkin
19 sudo systemctl daemon-reload
20 sudo systemctl start jenkins
21 sudo systemctl status jenkins
22 curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
23 sudo yum install unzip
24 sudo unzip awscliv2.zip
25 sudo ./aws/install
26
27

```

Figure A.14: Shell script used for installing softwares on AWS EC2

```

28 #ZAP is installed at /home/ec2-user/ZAP_2.11.1/zap.sh
29 sudo wget https://github.com/zaproxy/zaproxy/releases/download/v2.11.1/ZAP_2_11_1_unix.sh
30 sudo chmod +x ZAP_2_11_1_unix.sh
31 sudo ./ZAP_2_11_1_unix.sh -q
32 sudo tar -xvf ZAP_2.11.1_Linux.tar.gz
33 curl -o kubect1 https://s3.us-west-2.amazonaws.com/amazon-eks/1.23.7/2022-06-29/bin/linux/amd64/kubect1
34 chmod +x ./kubect1
35 mkdir -p $HOME/bin && cp ./kubect1 $HOME/bin/kubect1 && export PATH=$PATH:$HOME/bin
36 sudo cp kubect1 /usr/local/bin/
37 curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar."
38 sudo mv /tmp/eksctl /usr/local/bin
39 sudo yum install docker -y
40 sudo usermod -aG docker $USER
41 sudo newgrp docker
42 sudo usermod -aG docker jenkins
43 sudo newgrp docker
44 sudo service jenkins restart
45 sudo systemctl daemon-reload
46 sudo systemctl start docker
47 sudo systemctl enable docker
48 sudo systemctl status docker
49 sudo yum install jq -y
50

```

Figure A.15: Shell script used for installing softwares on AWS EC2 - cont'd