ANDRÉ EMANUEL
RAINHO BRÁS

SEGURANÇA DE CONTENTORES EM AMBIENTE DE DESENVOLVIMENTO CONTÍNUO

CONTAINER SECURITY IN CI/CD PIPELINES

Universidade de Aveiro
2021

ANDRÉ EMANUEL RAINHO BRÁS

SEGURANÇA DE CONTENTORES EM AMBIENTE DE DESENVOLVIMENTO CONTÍNUO

CONTAINER SECURITY IN CI/CD PIPELINES

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática e realizada sob a orientação científica do Doutor André Ventura da Cruz Marnõto Zúquete, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor João Paulo Silva Barraca, Professor auxiliar do Departamento Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho ao meus pais, às minhas irmãs, à minha mulher e ao meu filho pelo apoio incondicional.

o júri / the jury

presidente / president

Professor Doutor António Manuel Melo de Sousa Pereira
Professor Catedrático, Universidade de Aveiro

vogais / examiners committee

Professor Doutor Eduardo Resende Brandão Marques
Professor Auxiliar, Faculdade de Ciências - Universidade do Porto

Professor Doutor João Paulo Silva Barraca
Professor Auxiliar, Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço todo o apoio e acompanhamento dos orientadores desta dissertação, bem como todas as sugestões e paciência ao longo do processo.

Palavras Chave

Segurança de aplicações, Segurança de contentores, DevSecOps, SecDevOps, SLDC seguro, CI/CD, Docker, micro-serviços, Kubernetes.

Resumo

A ascensão da estratégia *DevOps* e a transição de uma economia de produto para uma economia de serviços conduziu a mudanças significativas no paradigma do ciclo de vida do desenvolvimento de software, entre as quais o abandono do modelo em cascata em favor de métodos ágeis. Uma vez que o *DevOps* é parte integrante de um método ágil, permite-nos monitorizar as versões actuais, recebendo feedback constante dos clientes, e melhorando as próximas versões de software. Apesar do seu extraordinário desenvolvimento, o *DevOps* ainda apresenta limitações relativas à segurança, que necessita de ser incluída nas *pipelines* de integração contínua ou implantação contínua (*CI/CD*) utilizadas no desenvolvimento de software.

A adopção em massa de serviços na nuvem e software aberto, a ampla difusão de contentores e respectiva orquestração bem como das arquitecturas de micro-serviços, quebraram assim todos os modelos convencionais de desenvolvimento de software. Devido a estas novas tecnologias, a preparação e expedição de novo software é hoje em dia feita em curtos períodos temporais e ficando disponível quase instantaneamente a utilizadores em todo o mundo. Face a estes fatores, a abordagem habitual que adiciona segurança ao final do ciclo de vida do desenvolvimento de software está a tornar-se obsoleta, sendo crucial adotar metodologias *DevSecOps* ou *SecDevOps*, injetando a segurança mais cedo nos processos de desenvolvimento de software e impedindo que defeitos ou problemas de segurança fluam para os ambientes de produção.

O objectivo desta dissertação é reduzir o impacto de vulnerabilidades em micro-serviços através do exame das respectivas imagens e contentores por um conjunto flexível e adaptável de ferramentas de análise que funcionam em *pipelines CI/CD* dedicadas. Esta abordagem pretende fornecer uma coleção limpa e segura de micro-serviços para posteriormente serem lançados em ambientes de produção na nuvem. Para atingir este objectivo, desenvolvemos uma solução que permite programar e orquestrar uma bateria de testes. Existe um formulário onde podemos seleccionar várias ferramentas de análise de segurança, e a solução executa este conjunto de testes de uma forma controlada de acordo com as dependências definidas. Para demonstrar a eficácia da solução, programamos um conjunto de testes para diferentes cenários, definindo as *pipelines* de análise de segurança para incorporar várias ferramentas. Finalmente, mostraremos ferramentas de segurança a funcionar localmente, que posteriormente integradas na nossa solução devolvem os mesmos resultados.

Keywords

Application security, Container security, DevSecOps, SecDevOps, Secure SLDC, CI/CD, Docker, microservices, Kubernetes.

Abstract

The rising of the DevOps *movement* and the transition from a product economy to a service economy drove significant changes in the software development life cycle paradigm, among which the dropping of the waterfall in favor of agile methods. Since DevOps is itself an agile method, it allows us to monitor current releases, receiving constant feedback from clients, and improving the next software releases. Despite its extraordinary development, DevOps still presents limitations concerning security, which needs to be included in the Continuous Integration or Continuous Deployment pipelines (CI/CD) used in software development.

The massive adoption of cloud services and open-source software, the widely spread containers and related orchestration, as well as microservice architectures, broke all conventional models of software development. Due to these new technologies, packaging and shipping new software is done in short periods nowadays and becomes almost instantly available to users worldwide. The usual approach to attach security at the end of the software development life cycle (SDLC) is now becoming obsolete, thus pushing the adoption of DevSecOps or SecDevOps, by injecting security into SDLC processes earlier and preventing security defects or issues from entering into production.

This dissertation aims to reduce the impact of microservices' vulnerabilities by examining the respective images and containers through a flexible and adaptable set of analysis tools running in dedicated CI/CD pipelines. This approach intends to provide a clean and secure collection of microservices for later release in cloud production environments. To achieve this purpose, we have developed a solution that allows programming and orchestrating a battery of tests. There is a form where we can select several security analysis tools, and the solution performs this set of tests in a controlled way according to the defined dependencies. To demonstrate the solution's effectiveness, we program a battery of tests for different scenarios, defining the security analysis pipeline to incorporate various tools. Finally, we will show security tools working locally, which subsequently integrated into our solution return the same results.

Contents

List of Figures

Acronyms

API	Application Programming Interface	1
AST	Application Security Testing	2
AWS	Amazon Web Services	7
CI/CD	Continuous Integration and Continuous Delivery	2
CI	Continuous Integration	20
CLI	Command-Line Interface	37
CM	Configuration Management	17
CaaS	Containers-as-a-Service	1
DAST	Dynamic Application Security Testing	2
DevOps	Development and Operations	33
GCP	Google Cloud Platform	7
GDPR	General Data Protection Regulation	15
IaC	Infrastructure as Code	8
JSON	JavaScript Object Notation	21
MSA	Microservices Architecture	1
OAS	OpenAPI Specification	55
OCI	Open Containers Initiative	11
OS	Operating System	10
PCI DSS	PCI Data Security Standard	15
PGP	Pretty Good Privacy	16
PR	Pull Request	46
PaaS	Platform-as-a-Service	7
PoC	Proof of Concept	5
QA	Quality Assurance	50
SAST	Static Application Security Testing	2
SDLC	Software Development Life Cycle	1
SSH	Secure Shell	16
TLS	Transport Layer Security	16
UI	User Interface	83
URL	Uniform Resource Locator	70
VCS	Version Control System	43
VM	Virtual Machine	4
YAML	YAML Ain't Markup Language	71

CHAPTER 1

Introduction

With the rising of the DevOps movement, the Software Development Life Cycle (SDLC) is changing from the waterfall model to agile models [14], allowing faster and at scale delivery of value and adaptation to market needs. New iterative and incremental [2] development methods benefit organizations by improving their business and increasing the quality of services delivered to clients. DevOps created or drove the use of new technologies such as cloud, containers, serverless, and open-source software in general. As cloud services and open-source software became widely adopted, monolithic applications tend to be replaced by microservices [16]. There is a tendency to migrate these applications to the cloud, and larger companies tend to move their applications from large virtual machines to several containers instead. Microservices' based applications frequently consist of clusters of hundreds of containerized services. These container clusters require availability, fault tolerance, and possibly geographically scattered. With the application's footprint increasing, container cluster management's complexity grows, and a new cloud service model called Containers-as-a-Service (CaaS) [27] emerges. CaaS providers deliver a container orchestration platform [9] to integrate and manage containers at scale. These platforms allow multiple provisioning and deployment options, such as auto-scaling, automated deployment [7] and can operate multiple containers as one entity for availability, scaling, and networking.

Microservices interact through Application Programming Interfaces (APIs) independent of the programming language and machine architecture. Therefore, surface exposure is distinct from the monolithic application's standard functions or routines, which only interact with other parts of the same application. In these container environments, microservices introduce new threat vectors. These can arise from images' integrity, how registries manage them, the level of isolation provided by orchestrators, vulnerabilities inside containers, and the operating systems that host the containers. Implementing microservices and their hosting containers expands the attack's surface in different directions, so it is vital to secure these services and correspondent containers before production for successful conformity and protection. Providing a secure Microservices Architecture (MSA) requires establishing a secure life cycle of containerized applications and controlling the interprocess communications of microservices. New best

practices for MSA security need to be adopted, such as encryption of all communications, requests' authentication, well-defined APIs and avoiding secrets inside the source code.

A novel challenge arises for traditional security methods that strive to manage these new threat vectors and related surroundings. Security operations potentially delay several times the application deployment in production environments. Furthermore, we may question: How can organizations ensure applications are trustworthy to release in production? How to follow security standards, compliance controls, regulations, and company policies without delaying development?

DevOps (see Figure 1.1) has different stages: plan, create, verify, package, release, configure, monitor. SDLC is a continuous cycle, applications are released daily or hourly, and fluctuations in the SDLC increase. Organizations acknowledge that DevOps' agility and speed bring new challenges, and securing the DevOps process is vital.

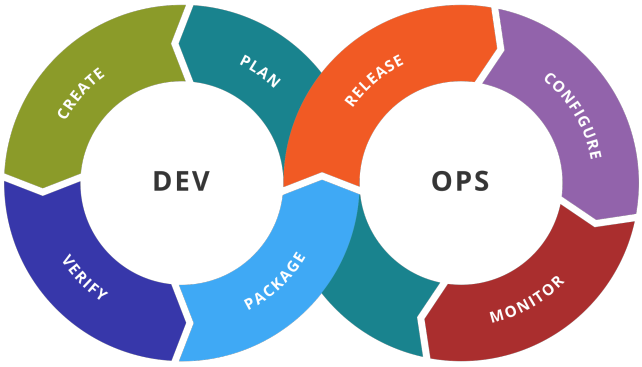


Figure 1.1: DevOps workflow. Image by Kharnagy [10]

With these challenges, an innovative movement called DevSecOps [15] appears. DevSecOps or SecDevOps [24] is the practice of including application security principles in a typical DevOps cycle, delivering security best practices earlier in SDLC. Reducing vulnerability impact in applications running inside containers will provide a clean and safe collection of microservices launched into production environments. Numerous vulnerability scanning tools can integrate into DevSecOps, for example, Static Application Security Testing (SAST) or Dynamic Application Security Testing (DAST) tools.

This thesis will study the integration of Application Security Testing (AST) in software development pipelines [11] by checking applications, their docker images, and related containers with Continuous Integration and Continuous Delivery (CI/CD) tools. We concentrate on the verify phase of the DevSecOps toolchain (see Figure 1.2). Our approach allows teams to apply various security checks on their CI/CD workflows while following organizations' policies. The pipelines will pass or fail, depending on the validation results of the security tools used. This will allow organizations to apply testing patches or remediation's during development and start injecting AST in the CI/CD pipeline.

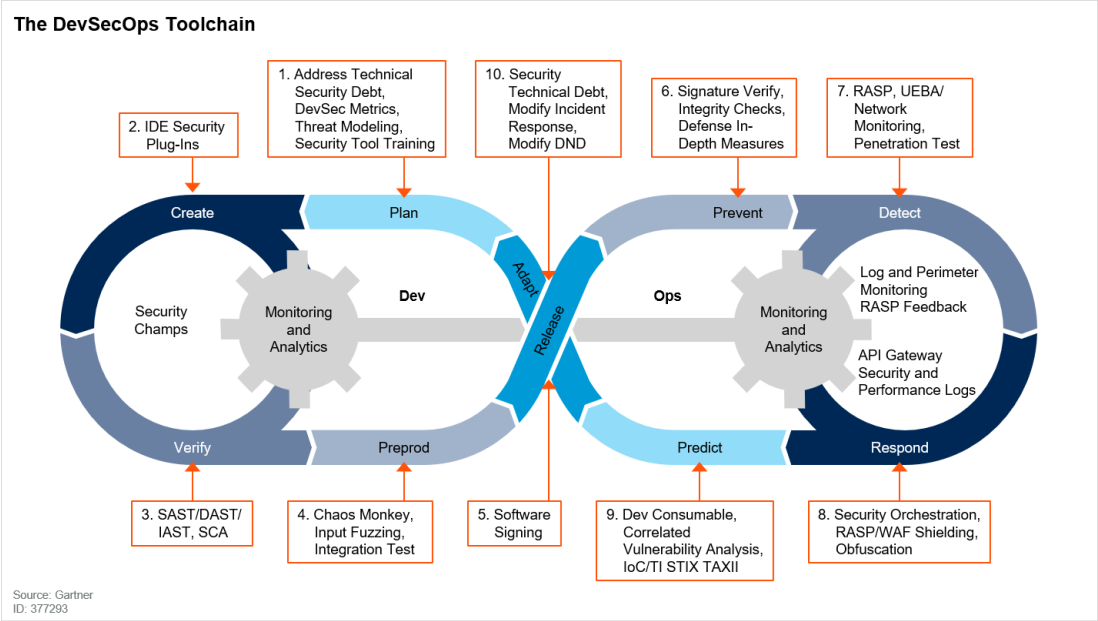


Figure 1.2: DevSecOps toolchain [8]

1.1 Problems of security within CI/CD pipelines

Organizations nowadays use microservices, agile models, and cloud computing [13], releasing software several times a day. Multidisciplinary teams participate in the software development of applications deployed and released online, maintaining and applying application security policies and mechanisms that warrant data privacy and client protection. Therefore, it reduces risk, protects the brand image by preventing leaks, builds and preserves customer confidence, and safeguards the integrity of the organizations' data and their customers' privacy. However, security is usually verified by internal security teams or external audits from third-party providers at the end of the SDLC, leading to delays in application release and reducing the frequency with which clients receive new releases. Companies involved in the fast-growing sector of online shopping already felt the need for security. For example, the payment card industry data security standard ensures that all entities that store, process, or transmit cardholder data maintain payment security [19]. Maintaining and applying security in applications will reduce risk, protect the brand image by preventing leaks, build and preserve customer confidence, and safeguard the integrity of organizations' data and the privacy of their customers.

Moreover, another problem arises if security checks are not applied in all versions of the applications that are released. If there are vulnerabilities in an application, this factor could lead to exposure of clients' private data, service unavailability, direct or indirect financial damage, and consequently a decrease in customer credibility. We could say that every team needs to address security, that it is everyone's job, and that security awareness workshops improve people's behavior. Still, such measures to avoid security issues are not enough. When it is everyone's job, very often, no one will take ownership of the security issues and related changes. Another important aspect is the error-prone human factor. One appropriate

solution is to apply security checks and validation as part of the software life cycle to improve application release speed. These security checks, validations, and tests need to be part of software pipelines to achieve it.

Alongside with embedded application Security in CI/CD pipelines in the SDLC, an effective security training could benefit organizations even more than just general security awareness workshops. Security training is a capability, a preventive control. Because of their knowledge and skills, developers are the right people for the job: Developers are familiar with the software and development teams of their organization. They have an in-depth understanding of the organization's technical issues and challenges. That is why hiring these developers as security champions [25] are advisable.

1.2 Research goals

Our work's main objective is to create a system for incorporating arbitrary safety tests in CI/CD pipelines. This work's motivation is reducing the occurrence of vulnerabilities in containerized application services launched in cloud production environments. Organizations, teams, and developers perceive the security risks inherent to new agile and fast software development practices using DevOps methodologies. Therefore, it is essential to support them by providing solutions and tools that adopt preventive software development practices. In this context, the idea of an integration solution appears. This solution will run several vulnerability analysis tools used in CI/CD pipelines during SDLC or by programmers locally on their machines. We need a flexible and adaptable solution that selects a set of analysis tools, runs them in dedicated CI/CD pipelines, and finally collects their results. We have developed a system that allows programming and orchestrating a battery of security tests. There is a form where we can select several security analysis tools and according to the defined dependencies, a set of tests is performed in a controlled way.

Our approach was creating different scenarios with applications in two different contexts. The two contexts are locally in a Virtual Machine (VM) and inside our solution. We make a validation with several vulnerability analysis tools and several vulnerable applications. Each vulnerable application is tested locally in an isolated environment, then inside our solution, and finally, we compare the results. We use vulnerability analysis tools such as secret detection, dependency scanning, static and dynamic application security testing for each vulnerable application for both contexts.

The obtained results demonstrate that inside our solution the analysis tools' effectiveness is not compromised. If a tool works in isolation, the same analysis tool will work in our solution's environment. We run these tools isolated inside virtual machines and later within the solution to demonstrate this. The tools' results remain unchanged whether they run within our solution or locally in virtual machines. Our solution runs the tools in isolated containers within a specific CI/CD pipeline that evaluates different security aspects selected and defined by developers and organizations. We define several scenarios that show that the

solution allows a controlled execution according to several dependencies. It is possible to orchestrate that a current test does not pass to a higher test if it fails or allows a particular test to fail. It is simple to add new tools to a battery of tests, integrating several analysis tools with different purposes.

1.3 Document structure

This document consists of seven chapters, the first of which is the present introduction. The following chapters are:

Chapter 2. Context. We discuss the technological context as the background for this work.

Chapter 3. Existing solutions. We present the security initiatives to add security testing in CI/CD pipelines, and analyze the available open-source and commercial solutions related to secret detection, static, dynamic analysis and integration solutions. We also approach the tools chosen for PoC scenarios and critical analysis.

Chapter 4. Architecture. In this chapter we determine the goals, analyze the usage scenarios and define the requirements to achieve the design of our security solution.

Chapter 5. Implementation. We explain the available choices, the decisions making, and how we implement the software components for our solution's Proof of Concept (PoC).

Chapter 6. Evaluation. We present the proof of concept scenarios, analyze if we fulfill the defined requirements, and summarize our solution's potentialities and results.

Chapter 7. Conclusion. Finally, we show the final notes about the work developed, issues raised, and future work possibilities.

CHAPTER 2

Context

In this chapter we discuss the technological context as the background for this work. The chapter is divided into four sections: Cloud computing, DevOps, Containers and CI/CD pipelines.

2.1 Cloud computing

Cloud computing is the on-demand provision of computer power, databases, storage, applications, and other computer resources via the Internet with charge-per-use. Cloud services platforms provide fast access to flexible and low-cost computing resources, whether running image-sharing applications to millions of users or supporting critical business operations [4].

This cloud computing concept, or *the internet operating system* as referenced by some authors such as Tim O'Reilly [17], provides access to infrastructure on-demand and at scale. Cloud computing started in 2006, when Amazon released Amazon Web Services (AWS). The architectural need dated to 2000, when Amazon's new business website service fought to become highly available and effective at scale [20] and therefore needed to increase computing resources and improve their strategies related to provisioning services automatically and expeditiously mainly because of Black Friday, Christmas, New Year and other special occasions. However, during the rest of the year, all infrastructure resources were underutilized. With this in mind, they tried to monetize the underutilized infrastructure and began providing enterprise IT infrastructure services as web services, now usually referred to as cloud computing[1].

In 2008, Google launched Google App Engine[2] that made it easy to build web applications, APIs and mobile backends at scale. This was the first service of the Google Cloud Platform (GCP). Google App Engine is a Platform-as-a-Service (PaaS)[3] that allows developers to build, host, and run their applications on Google Cloud. In 2010, Microsoft launched Windows

[1] https://aws.amazon.com/about-aws
[2] https://cloud.google.com/blog/products/gcp/your-favorite-languages-now-on-google-app-engine
[3] https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas

Azure platform[4], and the first services available were Windows Server in the cloud and SQL Azure hosted databases.

Benefits of cloud computing

Cloud computing provides access to infrastructure on-demand and at scale. Companies, especially start-ups, can benefit from this *"self-service infrastructure model"*. Programmable resources, dynamic capabilities, and a pay-as-you-go pricing model bring several advantages to business, like agility, elasticity, and cost savings. Companies can *"stop guessing about capacity"* and eliminate upfront investment in computing, storage, cooling, power supply, specialized staff, hardware, networking, or other IT infrastructure. Additionally, they can avoid wasting time and money with market predictions to estimate needs, acquire, upgrade, or retire IT infrastructure assets.

Operations and development departments can increase speed and agility to manage the infrastructure with programmable web services, treat Infrastructure as Code (IaC), roll-back application versions, use canary or blue/green deployment strategies, clone production environments for staging, create development environments and *"go global in minutes"* by provisioning applications in several regions worldwide in a short period.

Infrastructure-as-code and immutable infrastructure

In the shift from traditional infrastructures within a private data center to cloud systems, there are two main changes: first, the cloud ecosystem is primarily API driven, and second, there is a much broader elasticity of the infrastructure where, rather than months to years, it is now between hours to days the time a resource may exist. The need to bring the infrastructure up and down one time, a hundred times, or a thousand times appear. There is a shift in the fluidity of our infrastructure, and the cloud infrastructure is now disposable.

A question arises how to obtain the process guideline, code it, and ultimately automate it? If we define IaC[5], we can see an incremental history of who modified what and how infrastructure is specified currently in a version control system. Cloud infrastructure is managed as code, usually referred to as IaC. IaC speeds up infrastructure deployment and allows rapid iterations. It also eliminates non-standard configurations (snowflakes), builds consistency and repeatability to infrastructure provision and deployment processes. Examples of tools that enable IaC are Chef[6], Puppet[7], and, Ansible[8].

Being the notion of immutability is linked to the idea that once we create something, it no longer changes. Immutable infrastructure[9] follows that idea and is an approach for the provision, deployment, and maintenance of the infrastructure that never modifies servers

[4]urlhttps://docs.microsoft.com/en-us/archive/msdn-magazine/2010/january/cloud-patterns-designing-services-for-microsoft-azure

[5]https://www.hashicorp.com/resources/what-is-infrastructure-as-code

[6]https://www.chef.io

[7]https://puppet.com

[8]https://www.ansible.com

[9]https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure

after their deployment. On an immutable infrastructure approach, we never upgrade it on the spot. It means that if a server exists in version v1.0, we will not try to upgrade it to v1.1. We destroy the server from version 1.0 and create a new one based on a golden image tested and approved with version v1.1. This approach has positive security implications, because it is predictable and deterministic, avoiding occasional deviation or configuration drifts and removes administrative services such as SSH or FTP. Immutability has trade-offs if an application had a state and writes it to a local disk or volume. In these cases, we need to externalize the data so that the machine holding that application can be destroyed and created safely. Examples of tools that enable immutable infrastructure are CloudFormation[10] and Terraform[11].

2.2 DevOps

DevOps is a series of activities that simplify the processes between software production and IT teams to design, validate, and deliver software more quickly and efficiently [3]. The concept of DevOps is grounded on creating a collaborative culture among teams that were historically separated in their own silos. The rising of DevOps brings advantages like increasing trust, solving critical issues earlier, allowing faster software releases, providing users with new features more often, and enabling the proper management of unplanned work.

DevOps model defined

DevOps model[12] is a blend of philosophy, practices, and tools that improves organizations' ability to provide software and services faster. It is a collaborative culture with pillar ideas such as respect, trust, a healthy attitude about failure, and avoiding blame, as Flickr stated in 2009 on the *"10+ Deploys Per Day: Dev and Ops Cooperation"*[13]. presentation. Organizations that adopt DevOps model have delivery pipelines with common build, test, and release steps that serve the client and receive a feedback loop enabling them to monitor the current release and plan the next ones accordingly. In some DevOps models, security teams are becoming closer to, or even integrated, with development and operations' teams.

With the proliferation of public and private cloud platforms, such as AWS and OpenStack, there has been a trend in adopting disposable infrastructure instead of conventional static servers. There are distinct strategies to treat servers, choosing pets versus cattle, to manage their creation and destruction. One option is treating servers as pets [1], a common approach when working with legacy infrastructure. Another option is treating servers as cattle in elastic cloud infrastructure and emerging microservices. Cattle are servers destroyed after fulfilling their mission or having a problem, usually lasting one update iteration. Alternatively, pets

[10] https://aws.amazon.com/cloudformation
[11] https://www.terraform.io
[12] https://aws.amazon.com/devops/what-is-devops/
[13] urlhttps://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr?type=powerpoint

are servers that persist months or years of uptime and are fixed or serviced by operations teams. The cattle approach favors the stateless, microservice cloud-native methodology and the pet approaches any monolithic program, potentially an individual appliance.

The DevOps trinity

At its core, DevOps aims to align all life-cycle software development participants on three planes – people, processes, and tools – often referred to as the DevOps Trinity[14]. In this structure, the SDLC has considered two pieces to build a final construction; upstream and downstream, namely, development and operations. These two ingredients are components of the same software delivery process, although entirely decoupled in most organizations.

2.3 Containers

Containers are repeatable, self-contained execution environments, faster to wind up and down than virtual machines. They bring granular control over resources and lower footprint of each application in physical, virtual, or cloud hosts. They bring consistent environments, they run anywhere and provide some isolation. An application with dependencies and configuration with hooks into the Operating System (OS), can be packaged and shipped in a container.

Containers versus virtual machines

VMs share the same virtualized hardware resources within the hypervisor and run on top of it. Containers[15] uses a complete operating system (OS) with binaries, libraries, and application files. VMs consumes system resources and causes overhead when several VMs are running on the same physical server. In containers, there is no hardware emulation, which means a reduced penalty in terms of used resources. VMs can give us a consistent run-time environment and application sand-boxing, but containers bring extra benefits like small storage occupation, low overhead, increased portability, more consistent operation, and greater efficiency, providing better application development.

What problems can containers solve?

Containers can solve different problems, such as compatibility issues in application deployment, reduce resource footprint, impact on servers, bringing agility to software development. Containers isolate applications from each other unless they are explicitly linked. That means they avoid concerns about dependencies' conflicts or resource contention[16].

Containerized software runs reliably in different environments such as laptops, workstations, testing environments, or even production seamlessly. It has the ability to deploy applications

[14]https://www.cloudbees.com/blog/what-devops
[15]https://www.netapp.com/us/info/what-are-containers.aspx
[16]https://cloud.google.com/containers/

quickly, consistently, and smoothly. Containers minimize the impact of CPU, RAM, I/O, and networking consumption of a running application, since containers share binaries and libraries with other containers running in the same host. They can also help solve dependency loop problems, updated versions of an application, keep track of old versions, and roll back to other versions in a matter of seconds.

Container runtime

A container runtime[17] is a lower-level component typically used in a container engine but can also be used manually for testing. `runc` it is the most widely used container runtime that Docker, CRI-O, and other container engines rely upon. The Open Containers Initiative (OCI), created in June 2015 by Docker and other container industry leaders, presently holds two specifications: the runtime specification (runtime-spec) and image specification (image-spec)[18]. The OCI is an open governance structure for creating open industry standards around container formats and runtimes.

Container engine

A Container engine is a software portion that receives user requests, including command-line choices, pulling images, and operating the container from the end-user viewpoint[19]. There are several container engines, including Docker[20], CRI-O[21], RKT[22] and LXC[23]. In Docker, the Docker engine is responsible for distribution, orchestration, volumes, and networking. It has a daemon called `containerd` and incorporates the BuildKit[24], a Dockerfile-agnostic builder toolkit used in the image build process for Docker[25].

Microservices

One of the primary reasons for using containers is the growing adoption of microservices-based architectures. With microservices[26], it is possible to run multiple components of the same application on the same hardware independently, having much more control over the individual pieces and their life cycles. The microservices architecture is a series of independent services, each with its own unique purpose[27]. They are autonomous, specialized and typically interact over well-defined APIs.

[17] https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/
[18] https://opencontainers.org/
[19] https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/
[20] https://www.docker.com
[21] http://cri-o.io
[22] https://github.com/coreos/rkt
[23] https://linuxcontainers.org
[24] https://github.com/moby/buildkit
[25] https://docs.docker.com/engine/reference/commandline/build/
[26] https://www.redhat.com/en/topics/microservices/what-are-microservices
[27] https://www.docker.com/solutions/microservices

Microservices advantages

Therefore, we can ask why using microservices and if there are advantages involved. Each component service in a microservices architecture can be developed, deployed, and managed without affecting other service operations. Besides, each service is designed for a kit of capabilities and concentrates on solving a particular problem. This architecture provides a way to build microservices optimized for DevOps and CI/CD pipelines in the software development lifecycle. In short, microservices are highly sustainable and testable, loosely connected, individually deployable, built around company capabilities, and operated by a small team[28], and these are the advantages they can offer.

Container orchestration

Container orchestration is an automatic process for handling or scheduling individual containers' work for microservices' based applications across multiple clusters[29]. Containers changed the way organizations build, ship, and maintain applications. Essential aspects that lead to the rising of container orchestration were applications moving from monolithic to microservices, the mass adoption of containers[30], and the demand for automated ways to provision, schedule, and manage containers at scale. Mastering the lifecycle of containers, mainly in large and dynamic environments, has become vital. To maintain, operate, and automate many tasks, software teams use container orchestration[31]. Many container orchestration platforms are based on open-source versions like Kubernetes or Docker Swarm, but there are also commercial versions, such as Red Hat OpenShift, Amazon ECS, Azure AKS, among others.

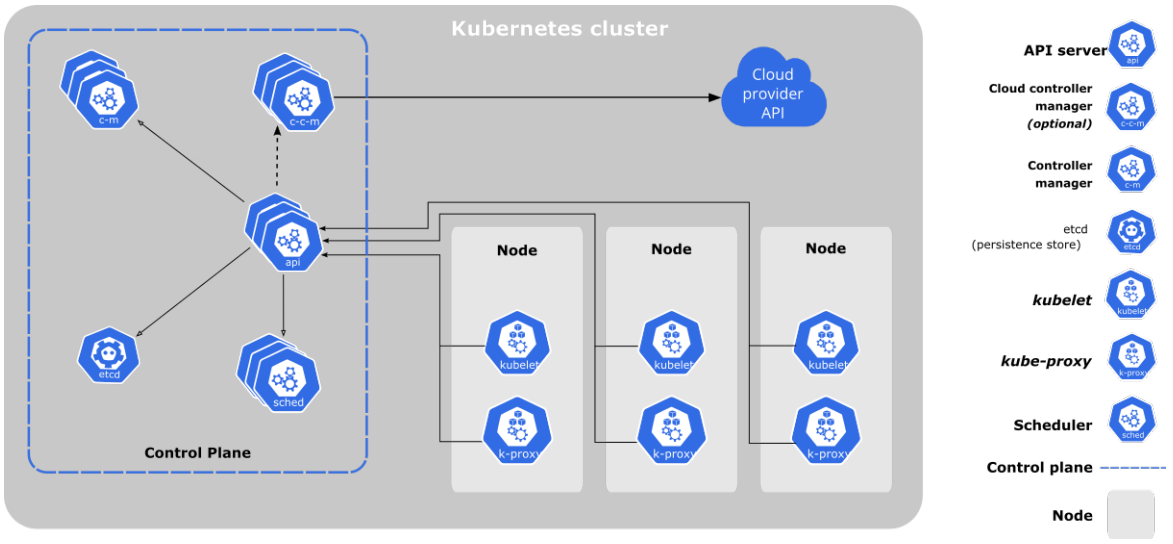


Figure 2.1: Kubernetes components. [28]

[28] https://microservices.io
[29] https://avinetworks.com/glossary/container-orchestration/
[30] https://sysdig.com/blog/sysdig-2019-container-usage-report/
[31] https://blog.newrelic.com/engineering/container-orchestration-explained/

Kubernetes

Kubernetes[32] is a lightweight, expandable, open-source platform for handling containerized workloads and resources. It has declarative configuration capabilities, providing manifests to define resources such as source code and consequent automation. Container orchestration platforms like Kubernetes can help build, test, monitor, and run the whole application lifecycle smoothly and efficiently. In Kubernetes, containers that are part of the same application form groups of logical units for easy management and service discovery.

Figure 2.1 shows a Kubernetes cluster[33] with all components connected. A Kubernetes cluster comprises one or more worker machines (nodes)[34] run containerized applications. The worker nodes host the Pods[35] that are the application workload components. The control plane[36] controls the worker nodes and resident Pods in a cluster. In production environments, the control plane typically operates through multiple machines, and the cluster generally runs various nodes, ensuring fault-tolerance and high availability.

The component `kube-apiserver` (api) is a part of the Kubernetes control plane that exposes the Kubernetes API. `etcd` is a reliable and highly-available key-value database used as backup store for all cluster data. `kube-scheduler` (scheduler) is a control plane component that tracks the recently created Pods with no designated node and selects the node to operate. `-controller-manager` (c-m) is a control plane component that handles controller processes. `cloud-controller-manager` (c-c-m) lets a cluster connect to the cloud provider's API and splits components that communicate with the cloud platform from those only interacting with the cluster.

2.4 CI/CD pipelines

A CI/CD pipeline[37] is a sequence of steps that must complete successfully to produce a new software release. Continuous integration or continuous delivery (CI/CD) pipelines are activities directed at optimizing software delivery through the DevOps or Site Reliability Engineering (SRE)[38] approach. The steps that generate a CI/CD pipeline are distinct tasks bundled in a pipeline stage. These tasks are commonly known as jobs. Software releases in the SDLC go through a series of typical stages, build, test, and deploy stages that are part of a CI/CD pipeline. A source code repository triggers a new pipeline cycle. A change in the source code will trigger a CI/CD tool notification, which executes the corresponding pipeline. Other common triggers include workflows that are automatically scheduled or started by the user. The application is compiled on the build stage, has the code tested by an automated process on the test stage, and finally deployed to production on the deploy stage. Notice

[32] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[33] https://kubernetes.io/docs/concepts/overview/components
[34] https://kubernetes.io/docs/concepts/architecture/nodes
[35] https://kubernetes.io/docs/concepts/workloads/pods
[36] https://kubernetes.io/docs/reference/glossary/#term-control-plane
[37] https://www.redhat.com/en/topics/devops/what-cicd-pipeline
[38] https://www.redhat.com/en/topics/devops/what-is-sre

that there is a previous deploy stage to a staging environment before the deploy stage to a production environment in most contexts. Other custom stages can exist in pipelines according to the organizations' constraints or different teams' specifics.

CI AND CD

A CI/CD pipeline automates software development by building code, running tests, and delivering new iterations of the application safely. These automatic pipelines remove manual errors, providing feedback loops to developers and enabling rapid product iterations. Continuous integration (CI) emphasizes testing automation to validate that the application does not break whenever new commits integrate into the main branch.

Continuous integration (CI) is a prerequisite of CI/CD, requiring that developers merge code changes to the master branch several times per day. For each code merged, an automated process triggers a new pipeline. First, building the application from the source code; then run a test sequence including unitary tests, user interface test, security test, or others; and finally create an artifact which can be a binary of the application, container image or website build.

The (CD) term could mean Continuous Delivery or Continuous Deployment. They are very similar, with only a significant difference in triggering application deployment, the first requires manual approval, and the second has an automated process. In Continuous Delivery, the code changes rolled continuously, although the deployments initiated manually. This means that every change in staging environments is proven to be deployable at any time on production since staging is nearly a replica of a production environment for software testing. Continuous Deployment goes beyond Continuous Delivery since every code change passing through all phases of a staging pipeline is released to production and disseminated to the customers without human intervention. Only a failed test will prevent a new change from being deployed in production.

CHAPTER 3

Existing solutions

The present chapter is dedicated to analyzing the existing solutions that verify security in applications, focusing on microservices deployed in Kubernetes in Docker run-times. We present the security initiatives to add security testing in CI/CD pipelines, and analyze the available open-source and commercial solutions related to secret detection, static, dynamic analysis and integration solutions. We also approach the tools chosen for PoC scenarios and critical analysis.

3.1 Security initiatives to add security testing in CI/CD pipelines

In the European Union, the General Data Protection Regulation (GDPR) [6] is applicable from 25 May 2018 in all member states. After that, many companies analyze how they can increase their applications' security level to follow GDPR obligations and avoid compliance issues. Security breaches can also have disastrous consequences on companies. For example, CodeSpaces - a code hosting and software collaboration platform - went out of business after a malicious party gained access to the company's AWS control panel [21]. Most of their data, configurations, and offsite backups were either partially or entirely deleted. With data breaches, GDPR regulation, PCI Data Security Standard (PCI DSS), other security standards or quality controls, building applications with security architecture design and built-in security in the development lifecycle became a vital piece for organizations.

Organizations aspire to deliver applications quickly and expeditiously, and a growing number of companies adopt DevOps methodology [18]. To follow agile development, comply with security standards and regulations, organizations must embrace new strategies to face these new challenges. Organizations start including AST in their CI/CD pipelines to ensure their applications are adequately tested before they reach production. This study [18] demonstrates the importance of including security validation in the CI/CD pipeline throughout the SDLC. In case of vulnerabilities in applications, attackers or malicious actors can access

private data or even secrets of users or companies. The same study analyzed nineteen developers of a specific company. These developers perform CI/CD pipeline tasks but rarely deal with security aspects. The research results show that CI/CD pipeline potentially presents high-risk vulnerabilities within applications, with non-encrypted connections outside the pipeline components and nonrestricted accesses.

Continuous integration and continuous delivery is a well-known activity in DevOps to ensure the quick delivery of new features. Standard security management strategies cannot stay current with this rapid life cycle of application development. Data breaches continuously compromise users' PII and secrets [12] by disclosing confidential information to unauthorized parties. Therefore, delivering high-quality protection for software systems has become progressively vital. DevSecOps attempts at incorporating security strategies into current DevOps activities. In particular, AST automation is a significant field of research in this phenomenon. According to T. Rangnau et al. [23], most of the current works incorporate only static code analysis and ignore dynamic test methods. This work [23] introduces alternatives to SAST integration techniques into a CI/CD pipeline, presenting approaches to integrate automated dynamic testing techniques into a CI/CD pipeline. DAST focuses on evaluating how a running application reacts to malicious requests. While significant, SAST cannot identify all security flaws in the system. In reality, static analysis can only discover vulnerabilities explicitly obtained from the source code, being a small subset of web applications' most common vulnerabilities. In contrast, with dynamic security testing, an application is targeted similarly to real-world environment attacks by hackers or malicious actors. Dynamic application security testing focuses on measuring how a running application reacts to malicious requests and capable of covering a much broader spectrum of flaws.

After the analysis of relevant works [12] [23] [5], we established that our solution would serve mainly to frame the execution of arbitrary tools in the AST area, such as: secret detection, static analysis, and dynamic analysis. We do not want to define a testing structure, but a way to frame several tests to be defined by the teams or individual developers in their CI/CD pipeline.

3.2 Secret detection

The secret concept is related to something that grants access to systems or enables authentication or authorization for humans or services. A secret is a piece of information only obtainable by one person or a group of individuals and not known by others. Secrets[1] can grant access to a system allowing users or services to authenticate or authorize themselves. Examples of secrets are passwords, usernames, access keys, API tokens, Pretty Good Privacy (PGP) or Secure Shell (SSH) keys, Transport Layer Security (TLS) certificates, or asymmetric credentials. Secret provides authentication of a system or service and authorizes someone to perform actions like reading, writing, or even destroying data.

[1] https://www.hashicorp.com/resources/what-is-secret-sprawl-why-is-it-harmful

The Uber data breach in 2016[2], disclosed the company's data on riders and drivers, including names, phone numbers, and email addresses, to unauthorized parties. After that event, organizations face new challenges concerning secret management to avoid the *"secret sprawl"*[3] phenomenon that has emerged when businesses and organizations migrate to the cloud. In addition to data breaches, organizations and users made public their source code repositories of software applications. These organizations or users send repositories to cloud hosting providers for collaborative software development using version control. The problem appears when this shared code is part of applications that need to handle credentials or secrets used for authentication, such as API keys, tokens, passwords, or certificates.

In [12] the authors analyze the rising problem of secret spread proliferation in GitHub. This work analyses secret leakage on GitHub by examining multiple files collected from public commits of repositories and a public snapshot of open-source repositories. This research reveals that information leaks impact over one hundred thousand repositories, and thousands of private secrets are disclosed daily. In [22] the authors present a tool for static analysis called Security Linter for Ansible and Chef scripts (SLAC). SLAC avoids secrets such as user names, passwords, and other hard-coded secrets embedded in the configuration files of Configuration Management (CM) tools. CM tools install and maintain applications on a server already in operation. While DevOps or operations teams are provisioning servers in cloud infrastructure or other systems at scale, handling CM tools could sometimes leave hard-coded secrets inside IaC config files. If this scenario occurs and secrets get disclosed, they could end up in malicious actors' hands, potentially employing them to exploit the provisioned systems. This work [22] intends to help practitioners avoid insecure coding practices while developing infrastructure as code.

Justification for having secrets inside source code can be an application or tool that needs to work off-line without external dependencies - a bank with data located in a particular country or region for legal motives. Several options are available to protect sensitive content, avoid plain-text credentials in playbooks, roles, configurations, or manifests, and accomplish these requirements. Organizations and their teams can use tools to encrypt secrets at rest in source code. Some examples are Mozilla SOPS[4] and Ansible Vault[5]. Isolation of secrets from source code is mandatory in specific organizations; there are tools to handle secrets for protecting sensitive data in transit and at rest. These tools can provide, audit, and revoke secrets at scale for services or even humans. Examples of these tools are HashiCorp Vault[6] and Google Berglas[7].

[2] https://www.nytimes.com/2017/11/21/technology/uber-hack.html
[3] https://www.hashicorp.com/resources/eliminating-secret-sprawl-in-the-cloud
[4] https://github.com/mozilla/sops
[5] https://docs.ansible.com/ansible/latest/user_guide/vault.html
[6] https://www.vaultproject.io/
[7] https://github.com/GoogleCloudPlatform/berglas

3.2.1 Secret detection - commercial solutions

Besides open-source secret detection tools, there are also commercial versions like AWS Secret Manager[8], Google Secret Manager[9] or CyberArk Secrets Manager[10].

`GitHub token scanning`[11] is written in Go[12] and based on the Hyperscan[13] library developed by Intel according to GitHub Blog[14]. The token scanning feature is part of the GitHub service that scans all changes to public repositories and public Gists[15] for credentials, but does not scan private repositories. Their goal is to identify secret tokens within the committed code in real-time and notify the service provider to take action. GitHub token scanning searches public repositories for established token formats to avoid malicious use of mistakenly committed credentials. When someone commits code to a public repository, or changes a private repository to public, GitHub scans the contents of commits in the repository for tokens belonging to a set of service providers: Amazon Web Services (AWS), Azure, Dropbox, Google Cloud, Slack, among others. If any set of credentials is detected, GitHub notifies the service provider who issued the token. After that, the service provider should revoke the token, issue a new one, or reach the developer directly.

`GitGuardian`[16] can clean up the version control system preventing secrets from entering the code base. This is a commercial solution to detect leaked sensitive information. They supply two different target audiences: developers and enterprises. It is free to use for individual developers and offers paid plans to protect organizations. They supply two different products, GitHub Public Monitoring (SaaS) that can monitor public repositories, and Private Repository Monitoring (On-premises) that can monitor private git repositories of organizations.

`Nightfall Radar`[17] Nightfall Radar is an API that uses machine learning to search GitHub repositories for confidential credentials and secrets, such as API keys for a wide variety of services (AWS, GCP, Twilio, Stripe, and more). The main features that distinguish this tool are that the exact types of keys or credentials do not need to be specified; all scans run asynchronous, and when completed, users are notified via email or webhook[18] endpoint; the results are available in a user interface. A `scan_id` is required to identify the scan and retrieve the scan results after authentication with the API is performed. Nightfall also states that it does not store or track sensitive findings. This tool uses deep learning methods to overcome the limitations of regular expression and Shannon entropy methods of other tools. Radar, an in-depth learning strategy, uses a model trained with features extracted from a broad set of API key patterns and their surrounding contexts in code.

[8] https://aws.amazon.com/secrets-manager/
[9] https://cloud.google.com/secret-manager
[10] https://www.cyberark.com/products/privileged-account-security-solution/application-access-manager/
[11] https://help.github.com/en/github/administering-a-repository/about-token-scanning
[12] https://golang.org/
[13] https://github.com/intel/hyperscan/releases/tag/v5.0.0
[14] https://github.blog/2018-10-17-behind-the-scenes-of-github-token-scanning/
[15] https://docs.github.com/en/github/writing-on-github/creating-gists
[16] https://www.gitguardian.com
[17] https://github.com/marketplace/watchtower-radar
[18] https://sendgrid.com/blog/whats-webhook

Shieldfy[19] discovers hard-coded secrets in repositories and notifies if it finds any secrets in the source code. It can detect embedded tokens, secret keys, and sensitive personally identifiable information (PII) in repositories to avoid accidentally including hard-coded secrets in the source code. Shieldfy can automatically detect and patch security problems and vulnerabilities in the source code before it goes into production. It has three main features: a quick code review, low false positives, and automatic fix and remediation. To obtain security feedback as soon as possible, it can analyze commits or even create pull requests. Another component is the Analysis Engine that extensively analyzes the meaning and flow of the source code to prevent false positives. Afterward, it prepares a patch for common vulnerabilities found and generates pull requests for each patch. The company shuts down Shieldfy[20] and open-sourced part of the written code on Github[21].

3.2.2 Secret detection - open-source solutions

Yelp's Secret Detector[22] enables to identify and avoid code's secrets. Yelp Engineering announced on their blog in 2018 that detect-secrets[23] tool was open-sourced to prevent secrets from being committed to the code base. It is written in Python[24], designed to be used as a git pre-commit hook[25], but it can also be invoked with scan parameters. This tool offers three components to set up: Client-side Pre-Commit Hook, Server-side Secret Scanning, and Secrets Baseline. Besides finding secrets in the source code, it has other capabilities like defining a baseline, accepting that there can be secrets in the source code currently, and, after that, auditing the baseline for changes. The server side of detect-secrets is called detect-secrets-server[26] that can track multiple repositories, periodically scan them and send alerts if it finds any secrets.

Repo Supervisor[27] checks security misconfigurations in source code, scanning for passwords and secrets. Auth0 engineering open-sourced Repo Supervisor[28]. This security tool provides secret detection in the source code, replacing the human factor as much as possible in the secure software development life cycle. It is written in JavaScript[29], and the main features that distinguish it from other tools are the ability to decrease the number of false positives, the capacity to scan data just for a file format, and serverless compatibility and webhook mode to check pull requests. It can provide alerts through Slack, slick HTML reports, and custom exclude lists.

[19] https://shieldfy.io/product/secrets-detection
[20] https://shieldfy.io/good-bye
[21] https://github.com/shieldfy
[22] https://engineeringblog.yelp.com/2018/06/yelps-secret-detector.html
[23] https://github.com/Yelp/detect-secrets
[24] https://www.python.org/
[25] https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
[26] https://github.com/Yelp/detect-secrets-server
[27] https://github.com/auth0/repo-supervisor
[28] https://auth0.engineering/detecting-secrets-in-source-code-bd63b0fe4921
[29] https://developer.mozilla.org/en-US/docs/Web/JavaScript

git-secrets[30] prevents organizations from introducing secrets and credentials into git repositories. Amazon Web Services - Labs created this tool that scans AWS credentials and other sensitive information in git repositories. AWS security best practices[31] advise people to use git-secrets when using a public Git repository for document or code versioning and sharing to avoid committing code or documents that contain sensitive information. This tool scans commits and commits' messages to prevent adding secrets into git repositories. If a commit or commit message matches one prohibited regular expression pattern, the commit is rejected.

truffleHog[32] explores git repositories, searching for secrets, investigating profoundly into the commit's history and across branches. This tool is advantageous in discovering unintentionally committed secrets. It scans every branch's entire commit history, examines each diff from each commit, and checks the secrets' existence. It works with regular expressions and entropy. For entropy tests, truffleHog measures the Shannon entropy for both the base64 charset and the hexadecimal charset for each text blob greater than 20 characters contained in each diff.

gitrob[33] is a recognition tool for GitHub organizations or users that helps identify potentially confidential files sent over to public repositories. This tool will clone repositories, iterate over the commit history, and flag files that match signatures on potentially sensitive files. It is written in JavaScript, and its main distinguishing features are configurable depth, a web interface that shows findings providing easy browsing and analysis, and the state of an assessment in memory, meaning that the results will be lost when Gitrob[34] finishes.

Gitleaks[35] examines the source code in git repositories for secrets. It offers a way to locate unencrypted secrets and other unauthorized forms of data in git repositories. This tool is written in Go, supports GitLab and GitHub with repository scans for bulk organization or repository owners (users), and scans requests for pull/merge in Continuous Integration (CI) workflow. It can audit uncommitted changes (pre-commit scans), uses the go-git[36] framework, enables environment-specific configuration, and has a JSON output format.

yara4pentesters[37] is a collection of rules for recognizing files containing juicy details such as usernames, passwords, and similar. This tool uses YARA, a sort of Swiss knife for malware researchers that does pattern matching. YARA is a mechanism that helps malware researchers recognize and classify malware. By creating a YARA rule, researchers can make textual or binary patterns consisting of a set of strings and boolean expressions that determine its logic. The yara4pentesters tool uses a YARA rule with patterns related to usernames, passwords, tokens, API keys, or other secrets instead of a malware family.

[30] https://github.com/awslabs/git-secrets
[31] https://aws.amazon.com/premiumsupport/knowledge-center/security-best-practices/
[32] https://github.com/dxa4481/truffleHog
[33] https://github.com/michenriksen/gitrob
[34] https://michenriksen.com/blog/gitrob-now-in-go/
[35] https://github.com/zricethezav/gitleaks
[36] https://github.com/src-d/go-git
[37] https://github.com/DiabloHorn/yara4pentesters

repo-security-scanner[38] is a command-line interface tool that detects secrets inadvertently committed to git repositories, such as passwords, tokens, private keys, and other secret. This tool, written in Go, helps developers that accidentally pushed sensitive data to GitHub to discover passwords, private keys, usernames, tokens, or other secrets. The only requirement for this tool is to download any target repository to investigate. It can examine the entire history of repository branches for secrets and add false positives to a file called `.secignore`.

repo-supervisor[39] examines source code for security misconfigurations, passwords, and other secrets. This serverless tool, written in JavaScript by Auth0[40] can scan all incoming pull requests in webhook mode[41]. It can scan data in a specific file format like JavaScript (*.js) or JavaScript Object Notation (JSON) files (*.json) and send Slack alert notifications if detected secrets. There is an excluded list exclude list, that can decrease the number of false positives. Without creating webhook, it is possible to use this tool if the source code is downloaded for later analysis locally. The tool reports are in HTML.

secret bridge[42] is a tool to help improve the ability to detect secrets exchanged on Github. Written in Python, this tool can operate in two modes: event polling or webhook. As soon as developers push new code to GitHub, events are received, and the script executes detectors to find leaked secrets. Three tools are supported: detect-secrets, git-secrets and trufflehog[43].

GitGot[44] is a semi-automated, feedback-driven tool to quickly scan for confidential secrets via public GitHub information. In Python, this tool empowers users to quickly search for sensitive secrets through public GitHub information. During scan sessions, users could provide input suggestions on ignoring the scan results, and GitGot will suppress them in the results' set. Users will blacklist files by file name, directory name, user name, or a fuzzy fit of the files' contents. Blacklists created from previous sessions can be saved and reused with related queries (e.g., company.com v.s. subdomain.company.com v.s. company.org). Sessions can even be paused and resumed at any time.

Ah, shhgit![45] searches secrets on GitHub in real-time. This tool, written in Go, finds secrets and sensitive files in real-time across GitHub code and Gist commits. It does this by listening to the GitHub Events API. It can also consume public APIs of GitLab and BitBucket. There is the possibility to use the tool locally and include it in CI pipelines to scan private repositories or other Git hosting services. This tool is distinct from other popular tools like gitrob and truggleHog, that concentrate on excavating through the repository history to discover secret files from organizations or users. The tool comes with one hundred and fifty signatures with the possibility to add more or remove any of them by editing a configuration file. Interesting features are increasing the number of concurrent threads or enabling the

[38] https://github.com/UKHomeOffice/repo-security-scanner
[39] https://github.com/auth0/repo-supervisor
[40] https://auth0.com
[41] https://auth0.engineering/detecting-secrets-in-source-code-bd63b0fe4921
[42] https://github.com/duo-labs/secret-bridge
[43] https://duo.com/labs/research/how-to-monitor-github-for-secrets
[44] https://github.com/BishopFox/GitGot
[45] https://github.com/eth0izzle/shhgit

search for high entropy strings.

3.3 Static application security testing (SAST)

Static application security testing[46] is a white-box testing methodology[47] that is performed with previous knowledge of the source code. It helps find security flaws[48] in applications without running the code. In our work, we will scan the application code itself and in particular cases Dockerfile or application dependencies. In [26] the authors stated that SAST helps discover software security flaws as early as possible. SAST is a focused security software test and assists in auditing the source code. According to the same source, software security is everyone's responsibility by enhancing the stability, consistency, and quality of code. Adding SAST into CI/CD pipelines adds value to developers by providing input about quality, style recommendations, and possible vulnerabilities and making code review more streamlined and transparent. For this reason, we considered the use of SAST tools in our security solution.

3.3.1 SAST - commercial solutions

`Black Duck`[49] by Synopsys focuses on reporting image inventory, mapping known security vulnerabilities to image indexes, container inventory, and cross-project risk reports. Multi-factor open-source discovery facilitates dependency analysis, file system scanning, snippet matching, and binary analysis, finding security risks, and suggesting a list of known fixes. It also analyzes the license risk, considering the containerized environment's software licenses bundled. Black Duck concentrates more on scanning and pre-production than on run-time security.

`Snyk`[50] has several security products: Snyk open-source, Snyk Code, Snyk Container, Synk infrastructure as code, an Intel Vulnerability Database, and License Compliance Management. Snyk enables organizations and developers to discover and address cloud-native applications' vulnerabilities. Kubernetes applications are not just container images; they also include prebuilt deployment models and default settings. Detecting vulnerable images in SDLC can be done locally with Snyk command line, on source control hosting providers, Docker registries, CI/CD pipelines, or even on Kubernetes clusters. There are two variants available, single testing or continuous monitoring. Snyk scans the base image for its dependencies of operating system packages installed by the package manager, libraries, and key binaries installed by other forms. After deployment, Snyk protects images for newly discovered vulnerabilities and Kubernetes applications for insecure configurations. Snyk focuses on detecting, fixing, and

[46] https://www.gartner.com/en/information-technology/glossary/static-application-security-testing-sast
[47] https://www.acunetix.com/blog/articles/dast-dynamic-application-security-testing
[48] https://www.owasp.org/index.php/Source_Code_Analysis_Tools
[49] https://www.blackducksoftware.com/
[50] https://snyk.io/product/container-vulnerability-management/

monitoring vulnerabilities in container images, and it can fix issues automatically, minimizing the exposure.

3.3.2 SAST - OPEN-SOURCE SOLUTIONS

Clair[51] is a project dedicated to static vulnerability analysis in the application container. It is open-source, currently including OCI and Docker and enabling clients to use the Clair API to catalog their container images and then match them to known vulnerabilities. Clair is a static vulnerability analysis tool for containers that collect vulnerability data, store them in a database, scan container images, and index the installed software packages. If any vulnerability matches the identified software packages in the image, it can send alerts, reports, or even block deployments.

Dagda[52] is a tool to do a static analysis of identified vulnerabilities, trojans, viruses, malware, and other malicious threats inside Docker images or containers. It can also monitor the docker daemon and running containers for abnormalities. Dagda retrieves information about the software installed on Docker images, such as the OS packages, programming language dependencies, and verifies if the version is free of vulnerabilities. It uses ClamAV[53], as an antivirus engine to detect trojans, viruses, and malware in docker images or containers, and it can also integrate with Falco[54]. Dagda supports multiple Linux-based images, like Red Hat/CentOS/Fedora, Debian/Ubuntu, OpenSUSE, and Alpine, and uses OWASP dependency check[55] for analyzing multiple dependencies.

Anchore Engine[56] helps developers to perform a thorough analysis of their container images, run queries, generate reports, determine CI/CD pipeline policies and correspondent behavior. It brings three core components to container certification: container inspection, container image analysis, and container policy evaluation. The key features are image analysis, policy management, notifications, CI/CD integration, Kubernetes integration, and orchestration. Organizations can submit an image to be analyzed, see if the images have any known vulnerabilities, evaluate the image against a predefined security policy, and set up a notification subscription when the image is updated. It has a Docker image available and can run in a CI/CD pipeline or within orchestration platforms, like Kubernetes. For CI/CD pipelines, there is a Jenkins plugin[57] available; however, there is also a standalone option available called anchore-cli[58] that runs on a command line. Anchore Engine is part of an open-source collection from Anchore that provides adequate tools for secure development. There is a toolbox made up of unique purpose tools for examining and scanning software projects among

[51] https://github.com/quay/clair
[52] https://github.com/eliasgranderubio/dagda
[53] https://www.clamav.net
[54] http://www.sysdig.org/falco
[55] https://github.com/jeremylong/DependencyCheck
[56] https://anchore.com/opensource
[57] https://plugins.jenkins.io/anchore-container-scanner
[58] https://github.com/anchore/anchore-cli

this collection. The toolbox is composed by syft[59] and Grype[60]. Syft produces a software bill of materials for container image or file system passed to Grype, which generates a comprehensive list of identified vulnerabilities within a container image or project directory. Grype find vulnerabilities for major operating system packages and vulnerabilities for language-specific packages.

3.4 Dynamic application security testing (DAST)

Dynamic application security testing[61] is a black-box testing methodology[62] performed without any knowledge of source code scanning for potential vulnerabilities in running applications. For our security solution, we considered their use on running containers.

3.4.1 DAST - commercial solutions

`Sysdig Secure`[63] uses a unified platform to deliver security, monitoring, and forensics in a container and has a microservices architecture. The key features are providing run-time detection, supporting incident response and forensics, data enrichment, image scanning, image auditing, and run-time vulnerability management. Scanning images can execute on the build process or directly on the container registry. It provides integration into CI/CD pipelines through an API or a Jenkins plugin. Sysdig Secure enforces compliance by preventing images with critical vulnerabilities from being uploaded into the image registry. Vulnerability information of package data comes from multiple sources. It is updated continuously from NIST Database to Official Debian, Ubuntu, RedHat, CentOS packages and security trackers, language-specific trackers, and other sources. Runtime protection will identify and block threats in real-time. This protection covers various aspects, tracking an entire application, container, host, and network system calls.

`Aquasec`[64] is a platform for container protection that allows maximum visibility to container operations, helping enterprises to identify and avoid malicious activities and possible attacks. It is a security suite designed for containers, replacing signature-based approaches and using machine-learned behavioral whitelisting, integrity control, and nano-segmentation. The key features are continuous image assurance, image-to-container drift prevention, enforcing least privilege, granular monitoring and logging, and container-level application firewall. This platform provides security audits, container image verification, run-time protection, automated policy learning or intrusion prevention. It offers an API and can be deployed on-premises or in the cloud.

[59] https://github.com/anchore/syft
[60] https://github.com/anchore/grype
[61] https://www.gartner.com/en/information-technology/glossary/dynamic-application-security-testing-dast
[62] https://www.veracode.com/security/dast-test
[63] https://docs.sysdig.com/en/sysdig-secure.html
[64] https://www.aquasec.com/use-cases/container-security

Tenable[65] facilitates DevOps processes seamlessly and safely by providing access to container images' protection – including bugs, ransomware, and policy breaches – through the integration into the building process. The key features are DevOps pipeline integration, in-depth visibility, automated inspection, continuous assessment, policy assurance, and run-time security. It is based on a technology called FlawCheck[66] that stores and scans container images. It can integrate with CI/CD systems that build container images to ensure security and compliance before containers reach production. The container run-time scanning detects new container images running in production, not tested for vulnerabilities and malware, assuring they are compliant with a policy.

NeuVector[67] is a security platform that provides vulnerability management during the entire CI/CD pipeline. It can assist across the whole container lifecycle, from image construction to container run-time and operation. The use cases are run-time protection, compliance, and audit. It can automatically discover the behavior of applications, containers, services and detect security escalations. It can monitor images in registries and admission control to block vulnerable images from flowing to production. The key features are image scanning, security auditing and compliance, integration with orchestration platforms and networking, network visibility and security, host and container monitoring, reporting and logging, and security auditing compliance. This product will secure the entire container stack in a container environment since it includes images, registries, containers, hosts, networking, and orchestrator.

3.4.2 DAST - OPEN-SOURCE SOLUTIONS

Falco[68] is an open-source project designed to detect intrusions and anomalies on native Cloud platforms such as Kubernetes, Mesosphere and Cloud Foundry. Falco may consume events from a variety of sources and apply rules to those events in order to recognize anomalous behaviour. Sysdig open-sourced Falco[69] that became a sandbox project within Cloud Native Computing Foundation (CNCF)[70] sandbox project. It has Integrations with Kubernetes, Mesosphere, Docker, rkt, among others. It enables organizations to gain insight into application and container behaviors. The key features are platform-aware, container-native, and deep visibility. It is a run-time security built for containers, based on prebuilt rules, allowing organizations to enforce policies across containerized applications and microservices.

OWASP ZAP[71] or Zed Attack Proxy (ZAP) from OWASP is an open-source Web application vulnerability scanner, which is an automated tool that scans Web applications from the outside

[65] https://www.tenable.com/products/tenable-io/container-security

[66] https://www.tenable.com/press-releases/tenable-network-security-acquires-container-security-company-flawcheck

[67] https://neuvector.com/container-security-platform

[68] https://falco.org/

[69] https://sysdig.com/opensource/falco

[70] https://cncf.io

[71] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

to look for security vulnerabilities such as cross-site scripting[72], SQL injection[73], command injection[74], path traversal[75] and insecure server configuration. ZAP also features an API that empowers developers to automate penetration testing and security regression testing within applications' CI/CD pipeline.

jaeles[76] is a tool for automated Web application testing that can orchestrate and integrate various Web scanners. This tool, written in Go, is a flexible and extensible framework for building a custom Web application scanner. Jaeles use signatures to identify threads written in YAML files. It includes a Web UI, REST API, and integrations with Burp Suite[77] and Osmedeus[78].

Nikto[79] is a Web server scanner that conducts detailed, multi-item Web server analyses. This tool, written in Perl, attempts to identify installed Web servers and related software. These checks will look for outdated versions and version-specific problems. It tries to recognize configured web servers and applications, scans server configuration objects such as various index directories and HTTP server options.

w3af[80] is an open-source web application security scanner. Writing in Python, this tool allows organizations and developers to detect vulnerabilities in their web applications. It has web and console user interfaces. Uses tactical exploitation techniques to discover new URLs and vulnerabilities. It supports different plugins such as brute-forcing, auditing, performing SQL injections, file inclusions, cross-site scripting, among others.

3.5 SAST or DAST with CI/CD integration solutions

In this section, we describe static or dynamic analysis tools that include CI/CD integration. GitHub, GitLab Secure, and Wallarm FAST are commercial solutions and huskyCI and Scan(skæn) are open-source solutions.

3.5.1 Commercial integration solutions

GitHub[81] enables organizations to secure their repositories[82] with features like features like secret scanning, dependency analysis and code scanning depending on the type of license

[72] https://www.owasp.org/index.php/Cross-site_scripting
[73] https://www.owasp.org/index.php/SQL_Injection
[74] https://www.owasp.org/index.php/Command_Injection
[75] https://www.owasp.org/index.php/Path_Traversal
[76] https://jaeles-project.github.io
[77] https://portswigger.net/burp
[78] https://j3ssie.github.io/Osmedeus
[79] https://cirt.net/Nikto2
[80] http://w3af.org/
[81] https://github.com
[82] https://docs.github.com/en/free-pro-team@latest/github/administering-a-repository/securing-your-repository

subscription. Free and paid GitHub products include alerts from Dependabot[83], a tool that creates pull requests to keep the code dependencies secure and up-to-date. GitHub sends Dependabot alerts upon detect any of the vulnerabilities in the GitHub Advisory Database affecting the packages that a repository depends on. GitHub provides code scanning, but is only available for organizations with an Advanced Security license. The GitHub One[84] product includes GitHub Enterprise, plus Advanced Security. Advanced Security includes code scanning and secret scanning (beta). Code scanning[85] is a feature that organizations use to analyze code in a GitHub repository to find security vulnerabilities and coding errors. Any problems identified by the analysis are shown in GitHub. GitHub One also brings the ability to use CodeQL code scanning with an existing CI system[86]

`GitLab Secure`[87] can perform static and dynamic analyses on application source code, verify for known vulnerabilities, and communicate them into merge requests, so that developers can correct them before the merge. GitLab Secure allows organizations and users to analyze various application security-related issues, such as container scanning, thread monitoring, dependency scanning, static application security testing, secret detection, dynamic application security testing, API fuzzing, among others. A security dashboard offers a high-level view of vulnerabilities found in groups, projects, and pipelines. The threat monitoring page presents security metrics of device environments run-time. Organizations and developers can identify risks and apply correspondent remediation. The types of tests related to application security are defined in YAML files in the application source code's repository. Organizations or users can select what type of tests they want to perform. Afterward, GitLab will run the correspondent CI/CD pipeline.

`Wallarm FAST`[88], or Wallarm Framework for Application Security Testing is a cloud-based application security framework conceived to help organizations automate protection and conduct security testing for microservices, websites, and APIs. It empowers businesses to reduce the time spent on security coverage significantly. Wallarm applies a DevSecOps approach by integrating tests in every build, using protections, and unifying CI workflows. It automatically transforms existing functional tests into safety tests and injects them in CI/CD pipelines. Wallarm FAST includes Wallarm Cloud with AI capabilities and a FAST proxy. The FAST proxy runs inside a Docker container that will capture requests and build baselines. It then generates and performs a variety of security checks for each build. It is possible to follow OWASP Top 10 standards or even specify testing policies list, such as payloads, fuzzer settings or parameters. FAST can report vulnerabilities or abnormalities into CI/CD pipelines and on ticket systems. Wallarm offers integration with various external platforms, including

[83] https://github.com/dependabot

[84] https://docs.github.com/en/free-pro-team@latest/github/getting-started-with-github/githubs-products#github-one

[85] https://docs.github.com/en/free-pro-team@latest/github/finding-security-vulnerabilities-and-errors-in-your-code

[86] https://docs.github.com/en/free-pro-team@latest/github/finding-security-vulnerabilities-and-errors-in-your-code/using-codeql-code-scanning-with-your-existing-ci-system

[87] https://docs.gitlab.com/ee/user/application_security

[88] https://wallarm.com/products/fast

Slack, Telegram, Jenkins, CircleCI, GitLab, and Selenium.

3.5.2 Open-source integration solutions

huskyCI[89] is a tool that can discover vulnerabilities within CI/CD pipelines by performing security testing inside them. It can integrate and execute various static security analysis tools. This tool combines support for languages like Python, Ruby, JavaScript, Go, Java, and Terraform HCL. It uses GitLeaks to perform secret detection on source code repositories, discovering passwords, API keys, tokens, and other hardcoded secrets. huskyCI is extensible since it enables organizations or users to add custom security tests into huskyCI. The API does the integration in CI/CD pipeline, and developers add a new job in a pipeline stage using the huskyCI client. The client sends requests to the API and obtains detailed data on issues. This data includes severity, file, line, and remediation advice so that developers can solve it.

Scan (skæn)[90] is an auditing tool for organizations and teams applying the DevOps approach. This tool discovers many forms of security vulnerabilities in applications and infrastructure code. A single search can retrieve results without having to connect to remote servers. Scan (skæn) has features optimized for DevSecOps workflow integration: overview feedback for pull requests, build breaker automation, GitHub code scanning, and Bitbucket code insights support. It was designed as a multi-scanner and incorporated various scanner tools with twelve possible CI/CD integrations, namely: GitLab CI, Jenkins, Circle CI, Travis CI, GitHub Actions, TeamCity and Bitbucket Pipelines, and among others. Scan has four main features, credentials scanning, static analysis security testing, open-source dependency audit, and license violation checks. The list of supported languages and frameworks has twenty-three different items. Programming languages include Java, Python, Go, JavaScript, and infrastructure as code includes Terraform, Ansible, Cloud formation, Kubernetes, and Serverless. There is an integration for Visual Studio Code IDE via an extension. With this extension, developers can conduct security audits and visualize the results inside the IDE.

3.6 Tools chosen for PoC scenarios

In this section, we indicate which tools we have chosen to test our solution and decision-making motivations. First, we looked for and filtered only those tools with an API or CLI integration. Secondly, we selected tools with a docker image maintained by their developers or recommended by them. Since our security analysis pipeline runs various analysis tools inside containers, this is an elimination factor. Third, to avoid licensing issues, inertia, delays, or limitations, we chose open-source tools. The last factor was finding tools adapted for CI/CD pipelines, with specific functionalities, such as returning a return state compatible with CI pipelines.

[89] https://huskyci.opensource.globo.com/
[90] https://slscan.io

For secret detection, we selected two candidates: truffleHog and shhgit. Both met the requirements, had CLI, had Docker image, and were open-source. Although shhgit can scan local repositories, there is no option to ignore paths and only return results in text or CSV format. It is more suitable for remote Git or Gist repositories. The final choice was the truffleHog because it is better suited for local repositories, an option to exclude a specific path and return results in JSON format.

In SAST, we selected two candidates: huskyCI and Scan (skæn). Both met the requirements, had CLI, had a Docker image, and were open-source. huskyCI is designed and adapted for CI/CD pipelines but has some drawbacks. It needs several components to work, such as an API node, MongoDB and PostgreSQL databases, and a frontend representing extra components to deploy and possible failure points. Scan (skæn) includes a docker image that runs standalone without additional dependencies. huskyCI has five supported languages and eight security tests available in contrast to the Scan (skæn) with twenty-three supported languages and frameworks. Both Scan (skæn) and huskyCI do dependency scanning and secret detection. Since Scan (skæn) supports more languages than huskyCI, it has more security tests related to dependency scanning. The final choice is Scan (skæn) due to the deployment simplicity and the broader range of languages and frameworks supported.

In DAST, we selected two candidates: W3af and ZAP. Both met the requirements, had CLI, had Docker image, and were open-source. ZAP has a Docker image called zap2docker with variants called stable, weekly, and live. These ZAP images' development is active and maintained, while W3af has a Docker image with five years old. We also found examples[91] of working integration with ZAP in GitLab CI pipeline. ZAP Docker image includes Python scripts dedicated to active, passive, and API scans with batteries of tests already configured. It is also possible to tune the image to fit and run inside CI/CD pipelines. The final choice is OWASP ZAP due to the simplicity and flexibility of Docker images and correspondent scripts included.

In terms of an external SAST, we choose a cloud platform called SonarCloud. This platform integrates with CI/CD systems, such as GitLab CI, using a docker image including a CLI tool called sonar-scanner. Sonar-scanner is part of SonarQube, an open-source platform developed by SonarSource for continuous inspection of code quality. Despite being a commercial platform, SonarCloud is free for open-source projects. SonarCloud enables us to visualize and analyze the code in a dashboard without deploying an instance of SonarQube. We choose SonarCloud due to CI integration, sonar scanner's facility, visualization simplicity, and local deployment absence.

3.7 Critical analysis

Our analysis focuses on the DevSecOps chain's verification phase, namely, application security testing. The application security testing has different approaches in the verification

[91] https://gitlab.tu-berlin.de/help/ci/examples/dast.md

phase: from SAST or DAST to interactive application security testing (IAST) and software composition analysis (SCA). In the various scenarios of our PoC, we only consider SAST and DAST from the verify phase of DevSecOps toolchain 1.2. We consider that the investigation's software target already includes security concerns since its design and security faults or flaws are outside our research scope. We build our solution to find coding errors or implementation problems (bugs) and not design errors (faults). We intend to include vulnerability testing to verify security issues in the software development to ensure that the applications that go into production will not break the purpose of these applications' services. The aim is to maintain a continuous delivery integrated with production, avoiding disruptive processes.

Our main work objective is to create an integration solution that incorporates arbitrary security tests on CI/CD pipelines. We want to give organizations the possibility to use several CI/CD systems without being restricted to only one. We want developers to get results from vulnerability analysis tools locally on their machines during the SDLC. We only used containers for running the security analysis tools inside our solution. It should be possible to run in VMs but our analysis did not measure them up. Our solution will use cross-sectional vulnerability analysis tools used in CI/CD pipelines or developers. The idea is to encode a battery of tests in a form and place it along with the application's source code. This form, which is sent to our solution, triggers automated tests, including several tools for detecting vulnerabilities. These security tests will run in a CI/CD pipeline after the functional tests pass.

To fulfill these purposes, we need solutions that allow this flexibility. HuskyCI and Scan(skæn) are open-source solutions that integrate several static code analysis tools. Both allow dealing with false positives and ignoring specific vulnerabilities. However, they do not allow the integration of new tools or change test execution order. These two features are central to our solution, and these two solutions are discarded. Wallarm FAST is a DAST plus fuzzing with API and CI/CD systems integration but lacks flexibility. We cannot add new analysis tools or define execution order, so Wallarm was excluded.

GitHub enables integrating new security tools with workflow templates in GitHub Actions. The templates defined as code adds popular external services or setup new workflow with custom tools. GitLab Secure enables integrating a diversity of existing and preconfigured security testing tools and run them in GitLab CI. It also has code templates for the various security tests. It is also possible to define new templates and add custom tools with specific tests. GitHub and GitLab Secure both provide ways to define workflows or pipelines as code in YAML files. It's possible to run particular jobs within Docker containers. Both GitHub and GitLab Secure have APIs to create workflows or pipelines. However, they are not straightforward and require time and effort to integrate them on other CI/CD systems or by developers on their machines. Although GitHub and GitLab Secure provide flexibility to add new tools, organizations need to migrate all their source code to these platforms with all the privacy implications.

The static or dynamic analysis tools described above 3.5 include integration into pipelines CI/CD. Still, some lack flexibility, others simplicity and privacy. Our work differs from others

for four main reasons. First, it provides independence from the CI/CD system type. The second is simple to use in a CI workflow or on a programmer's console machine. The third can work alone in a cluster or private environment and finally is open-source.

CHAPTER 4

Architecture

This chapter describes the work carried out in this thesis, namely, the design and specification of what we call SecureApps@CI solution. This is meant to replace the existing security testing applied at the end of SDLC. The new capabilities activated from beginning to end and across SDLC will allow an earlier discovery and reduction of vulnerabilities, thus effectively building a secure SDLC. The SecureApps@CI solution complements the work already done by internal security teams and external security audits to provide precise and valuable data for PCI audits[1].

SecureApps@CI achieves three relevant improvements regarding the usual tests executed manually by internal security teams or external auditors or companies at the end of the SDLC. The first improvement is putting secret detection in place for the source code repositories at every CI/CD pipeline run; the second is adding automated static code analysis to pipelines; and the third is injecting automated vulnerability scans along the SDLC. Usually developers, Development and Operations (DevOps) or even operations (Ops) teams perform ad-hoc secret detection, quality assurance (QA) teams perform manual code review and security (Sec) teams perform manual penetration testing. With this in mind, SecureApps@CI is born as an integration solution that can launch several security tools, run them concurrently or sequentially, and filter or aggregate their results. Finally, this solution launches a stack of security tests that different teams can customize, thus, developers, operations, as well as security teams are able to reduce risk, build customer trust, protect brand image, and safeguard their organization's valuable assets and their customers' data privacy.

4.1 Goal

The main goal of the SecureApps@CI solution is to ensure that containerized applications have no dangerous components, by providing a sanitizing process that guarantees these applications will run free of vulnerabilities at the moment of their deployment. In the

[1] https://reciprocitylabs.com/what-is-a-pci-audit

SDLC process, applications developed and released from a CI/CD pipeline generally do not have a quality process focusing on security elements. During the development phase, QA frequently links to user experience, usability, portability, or end-user experience activities to ensure good interaction between the end-user (client) and the service that the application itself provides. SecureApps@CI can tackle security issues from the beginning of the SDLC process by analyzing three default test vectors: secret detection, static analysis, and dynamic analysis. However, it can also analyze other custom vectors like license violation checks or open-source dependency audits. Customized test vectors will need custom security tools added to SecureApps@CI. We start by presenting the various SecureApps@CI scenarios, next we enumerate the requirements, and then we move to its architecture. Lastly, we will approach the design of the SecureApps@CI that is relevant to our PoC implementation and the results.

4.2 Scenarios

SecureApps@CI has two distinct scenarios in its foundation. As shown in Figure 4.1, it can be called and used by pipelines or developers during the SDLC. CI/CD systems will call SecureApps@CI API within CI/CD pipeline and developers will call it from their laptops or workstations. Both options use a distributed version control system to work. On CI/CD systems, the pipeline will run every time there are code changes. The SecureApps@CI API can be requested in stages build, test, deploy, or other desired. A new security analysis will run according to a configuration file. Developers download code from a source code repository, make changes to one or more files within the code, send it over to the remote repository, trigger an automated action that will call the SecureApps@CI API, and a new security analysis will be executed.

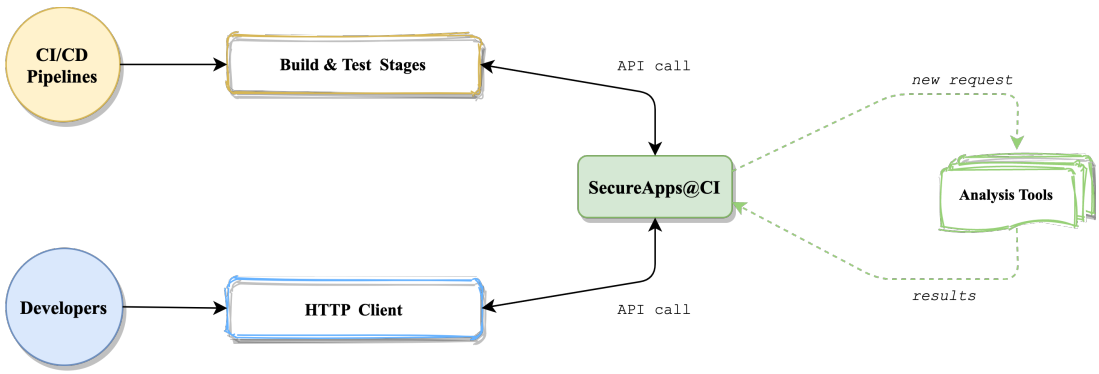


Figure 4.1: Overview of scenarios

4.2.1 Scenario1 - inside the pipeline

As shown in Figure 4.2 the SecureApps@CI can be used by CI/CD pipelines or other services running on any machine as long as it has an HTTP client installed. Let us consider a

common application pipeline that starts with the build stage. It has build, test, and deploy stages and on the test stage an API call is executed. This call sends a security analysis definition to the SecureApps@CI. It triggers a new security analysis behind the scenes, runs the correspondent security tools for each job defined in the security analysis definition received and returns `pass` or `fail` depending on the sum of the security tools results. If all security tools run without finding any security issues or the ones found are white-listed, the pipeline will move on to the deploy stage and reaches the end. If security issues are found, no white list is in place, the pipeline will fail and end.

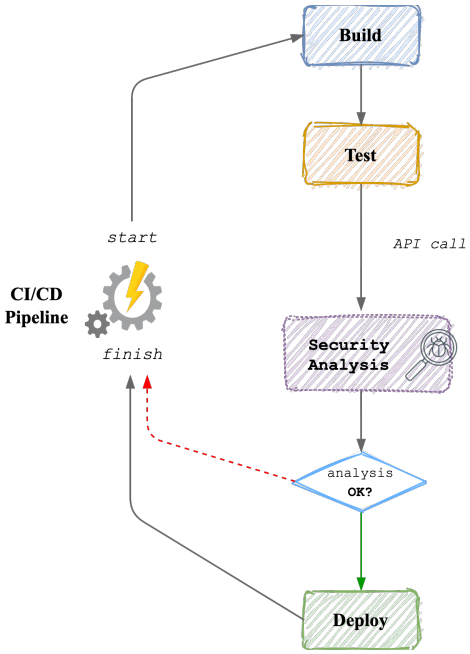


Figure 4.2: Scenario1 - inside the pipeline

Developers, operations, or security teams can write a definition file and have a security analysis running in their application pipeline, turning the security into a *"low hanging fruit"* close to everyone inside these teams. Multidisciplinary teams like DevOps, SecDevOps, or DevSecOps teams most likely will have an important role in the SecureApps@CI implementation, maintenance, and improvement. This can go even further and facilitate work and interaction between teams around the security topic, which is often viewed as a major blocker in application development. The success in the adoption of SecureApps@CI, the security analysis, usage, and correspondent application sanitization and cleaning process will benefit every team and the organization itself.

4.2.2 Scenario2 - the developer side

As shown in Figure 4.3, the SecureApps@CI can be used by the developer as long as they have an operation system with an HTTP client installed. Let us consider a developer

that creates new code, and afterward an API call is triggered by a hook[2] or script. This call sends a security analysis definition to the SecureApps@CI. The solution triggers a new security analysis behind the scenes, runs the correspondent security tools for each job defined in the security analysis definition received, returns `pass` or `fail` depending on the sum of the security tools results. If all security tools run without finding security issues or they are white-listed, the code will be sent to a VCS and a new application pipeline is launched. If any security issues are found, and no white list is in place, then the code will need a review from the developers or team involved in the SDLC of the corresponding application.

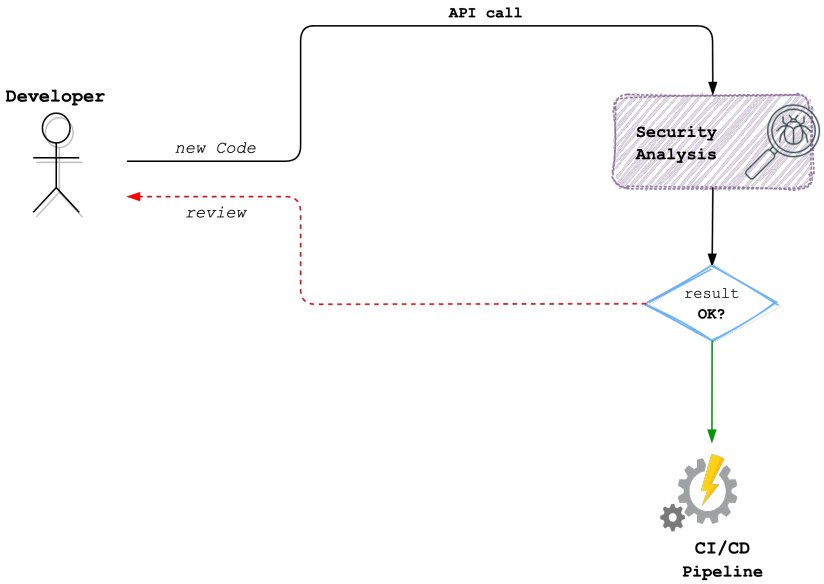


Figure 4.3: Scenario2 - the developer side

This scenario is particularly useful for teams during SDLC, when developers are implementing new functionalities on specific branches of applications, and afterward merging code to master when the functionalities are finally working. During this process, if they have the ability to launch a security analysis of the SecureApps@CI with a simple API call, checking the security issues and solving them before the application pipeline runs, this will avoid the burden of waiting for the pipeline to finish to have the security analysis' results.

This approach reduces the inertia of using the SecureApps@CI by decreasing the development time, while avoiding potential stress or demotivation in using it. Since pipelines typically have several stages including a test stage in the middle or in the end, if we add more analysis of security tools, we will be creating obstacles to the development flow. For these reasons, we expect programmers to have the best possible interaction with the solution, increasing confidence, comfort, and motivation to continue using it.

[2]https://www.atlassian.com/git/tutorials/git-hooks

4.3 Requirements

Given our main goal presented in Section 4.1 and having a CI/CD pipeline of applications in mind, the solution's design has the purpose of achieving a functional security analysis solution. For the PoC we need flexible and integration capabilities, with developers and CI/CD pipelines being able to interact with the system during the security analysis. To achieve this, the SecureApps@CI must meet the following points.

Analysis definition

The definition of the security analysis pipeline is declared in a document which can be a form. This document should have both a human and machine readable format and is sent over to the SecureApps@CI as a file. Only one file format is allowed. In this document, it is necessary to choose the application to be analyzed and correspondent details: the branch, environment, source code repository, and team. There is also the possibility to add more parameters to determine how, when, and where the tools will run. Special conditions such as outbuildings or white lists can also be defined in the document.

Components

All components and pieces in the SecureApps@CI will be leveraged by open-source software. The security tools built, analyzed, and tested by security analysis pipelines are just open-source tools with a Command-Line Interface (CLI) or an API integration. Commercial security tools can also be integrated, if adjusted and tested, but are out of the scope of PoC implementation.

Deployment

The deployment of all components in the SecureApps@CI uses a container engine to launch them. The construction of components employs minimal container images to achieve this requirement, including just applications and dependencies. Before each application runs inside a container, a static analysis of dependencies is performed to ensure we do not run any vulnerable dependencies. After a successful build of components, a container engine executes their deployment.

Architecture

The architecture should be modular and extensible. Each block has a focused functionality, consistent interfaces, and contains one or more components to fulfill this requirement. The blocks that have boundaries with others and provide a service will expose it through an API. This way, we ensure that blocks interacting and communicating with others can do it consistently. The exceptions are the component that shows the results and the agent nodes that run the security analysis. This way, the solution gains interoperability and provides the flexibility to attach or detach new components inside each block if necessary.

Availability

The SecureApps@CI must be available for CI/CD pipelines or developers without plugins or additional packages. To reach this requirement, we created an API that works as an interface between the developers or CI/CD pipelines and SecureApps@CI. This way they can create a security analysis of an application just by making a request from an HTTP client inside a source code repository.

Results and report

The analysis results include a Boolean output (pass or fail) and a detailed report that each security tool provides. The report format depends on the security tool output; however, whenever possible, we will prefer light data exchange formats, simple to read by humans and easy to process by machines.

Web interface

The provided analysis results - the Boolean output, and a report - are visible through a web interface, showing if the analysis passed or failed, and the detailed report output of each security tool.

Distributed version control system

Git is the chosen distributed version control system. It is a strict requirement given the context of SecureApps@CI. The environment of organizations in the context of our analysis, and the consequent usage of CI/CD pipelines employ Git as the only distributed version control system.

Client requirements

Clients of the solution must have an HTTP client. The potential clients of SecureApps@CI, namely, autonomous CI/CD pipelines or machines operated by developers, need an HTTP client installed to request new security analysis pipelines. The source code of the applications to be analyzed must be in a Git repository, as stated in the previous requirement. By reducing the client requirements to a bare minimum, we intend to improve compatibility and consequently increase the adoption of SecureApps@CI.

4.4 Architecture

After studying and analyzing the SecureApps@CI requirements, we started modeling and choosing the building blocks needed to satisfy and fulfill the requirements. As shown in Figure 4.4, an architecture model emerged, having blocks providing a particular service to

others. Our model's building blocks are a receiver, a nucleus, an orchestrator, and a visualizer. The receiver is an interface between clients and the SecureApps@CI. It listens for new security analysis requests and receives a definition document from clients. Next, the nucleus manages security analysis requests according to this definition, acting as a verifier of requests, a filter of definition requirements, and a processor of actions to execute. Afterward, the orchestrator works as a dispatcher launching agent nodes to execute security tools according to the analysis. All jobs will run inside containers, in series or in parallel depending on the analysis definition document. The security tools 1, 2, and n presented in the figure represent the possibility of launching several security tools for each analysis. In each analysis, a nucleus will verify dependencies, calculate logical operations between security tools, and aggregate each outcome to produce the final results. Finally, the receiver will return a Boolean result to CI/CD pipeline and the visualizer will show the results and detailed reports to the developers in a dashboard. The visualizer may show the execution flow of the safety analysis pipelines; however, our key foci are the results and reports.

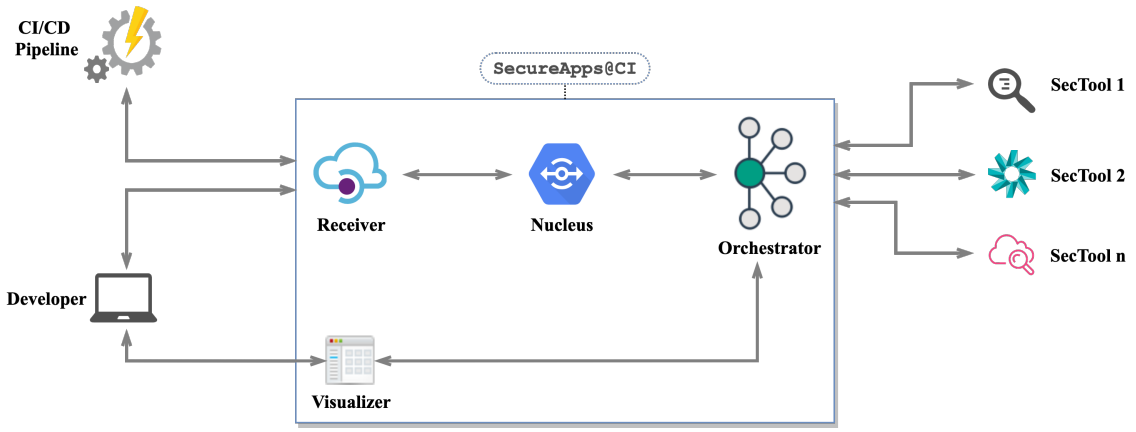


Figure 4.4: Architecture of SecureApps@CI and its integration with CI/CD pipelines, developers and arbitrary, external security analysis tools

4.5 Solution design

In this section, we will proceed with the design of SecureApps@CI solution. We start by identifying the communications between the blocks of architecture. Next, we will define the solution integration, the type of services used, the procedures, and the type of results of SecureApps@CI analysis. Finally, we describe the security analysis flow. Notice that different security tools can run on each analysis, such as secret detection, dependency issues, static or dynamic analysis, external analysis, among others.

For example, a typical use case has the first stage mandatory (e.g., secret detection). If there is a successful execution, the analysis moves to stage two. Secret detection fits in stage one to ensure that organizations do not expose secrets, API keys, tokens, passwords, or credentials to the outside world. After this, in stage two, a static analysis is done to

verify the application code, configuration files, container related files, and dynamic analysis to check vulnerabilities in running containers. Jobs can run parallel on stage one, for example, two secret detection tools. Nevertheless, also in stage two, for example, static and dynamic analysis can run simultaneously.

Architecture and communications

The three main components of the SecureApps@CI solution are the **receiver**, **nucleus**, and **orchestrator**. Each will provide a service to the left block of the architecture model referenced in Figure 4.4. We follow the principles of a microservices' architecture, so we designed each component as software application suites of independently deployable, loosely coupled, collaborating services. These services have their stack, including the database if applicable, and will communicate with one another over APIs. The use of APIs enables and promotes faster dissemination of new functionalities and updates. Replacing or improving services can be done without affecting other architecture services. The exception is the **visualizer** that is not designed as a service and will not provide any consumer API. The **visualizer** depends on the selected **orchestrator** since our system can use several orchestrators. We do not know their ability to provide execution flows or be easily generalizable. It is not a priority to visualize the executed flows but to ensure that the applications are clean, safe, and without vulnerabilities. Therefore we have not defined an API for the **visualizer**.

Next, we will specify the messages between the various parts of the architecture and describe the individual requests and responses from a high-level perspective. The first part is to verify the authorization, namely, if the `api_key` on the request of a new security analysis is valid. If the `api_key` in the request matches the configured key, the request is authorized. The diagram in Figure 4.5, shows a message sequence chart with a failed authorization. A **client** sends a security analysis request to the **receiver**, an authorization request is passed to the **nucleus** that verifies the `api_key` and declares unauthorized since the key is not correct. The **client** receives a response stating that the request is unauthorized and the security analysis will not execute.

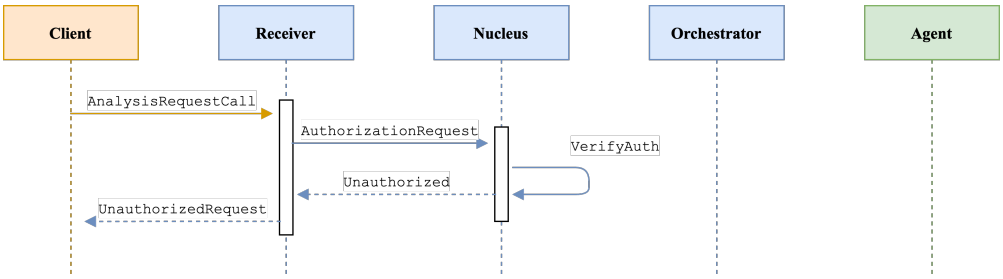


Figure 4.5: Message sequence chart - authorization checking failure

After the authorization request, the request for a new security analysis needs to be validated. For this, the security analysis definition document will be parsed and passed through a linter[3]

[3]https://en.wikipedia.org/wiki/Lint_(software)

to signal syntax or stylistic errors, invalid parameters and suspicious constructions. The **orchestrator** performs the validation since it is aware of the specifics and constraints of the definition document must have. It receives security analysis definitions and schedules the security tools jobs accordingly to agents. The diagram in Figure 4.6 depicts a message sequence chart with failed validation, which means that the security analysis will not occur. A **client** sends a security analysis request to the **receiver** and an authorization request is passed to the **nucleus** that verifies if the `api_key` is authorized to make the request. If the authorization is valid, a validation request sent to the **orchestrator** will verify the request. In this case, the request is invalid, the verification fails, and **nucleus** forwards the response to the **receiver**. Finally, the **client** receives a response stating an `invalid request` and the security analysis will not occur.

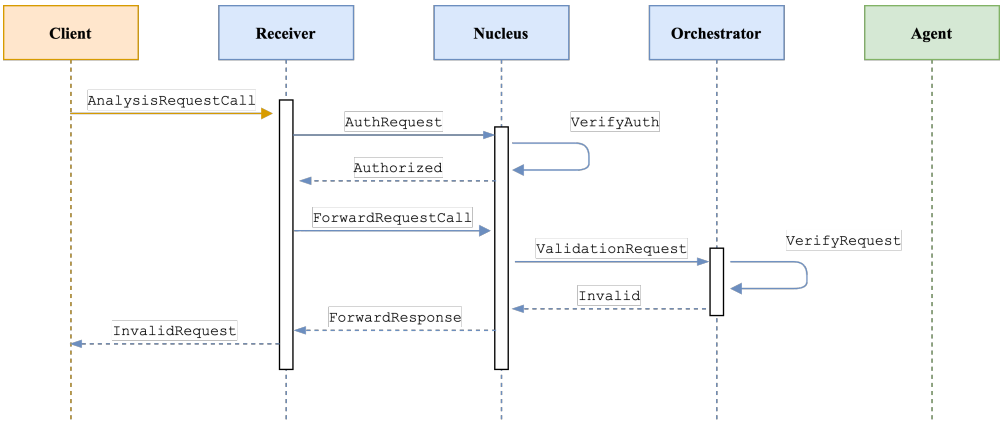


Figure 4.6: Message sequence chart - analysis document validation failed

Once the request has passed the authorization and validation stages, it starts to create an analysis call to the **orchestrator**. If a new security analysis is accepted, the **orchestrator** will launch all analysis jobs. Each job is composed of one or more security analysis tools that will verify applications' security issues and vulnerabilities. These security tools which create results, the **orchestrator** collects them and forwards the raw results to the **nucleus**. The **nucleus** filters the results, sends a response with filtered results to the **receiver** that passes it to **clients**. The diagram in Figure 4.7 is a message sequence chart with a valid request, meaning the security analysis will occur. The **nucleus** verifies and accepts the authorization, the **orchestrator** validates the request, creates a new analysis, and correspondent analysis tools are launched. The results of the analysis tools are retrieved properly: the **orchestrator** routes the results to the **nucleus**, which filters and sends them back in response to the **clients**.

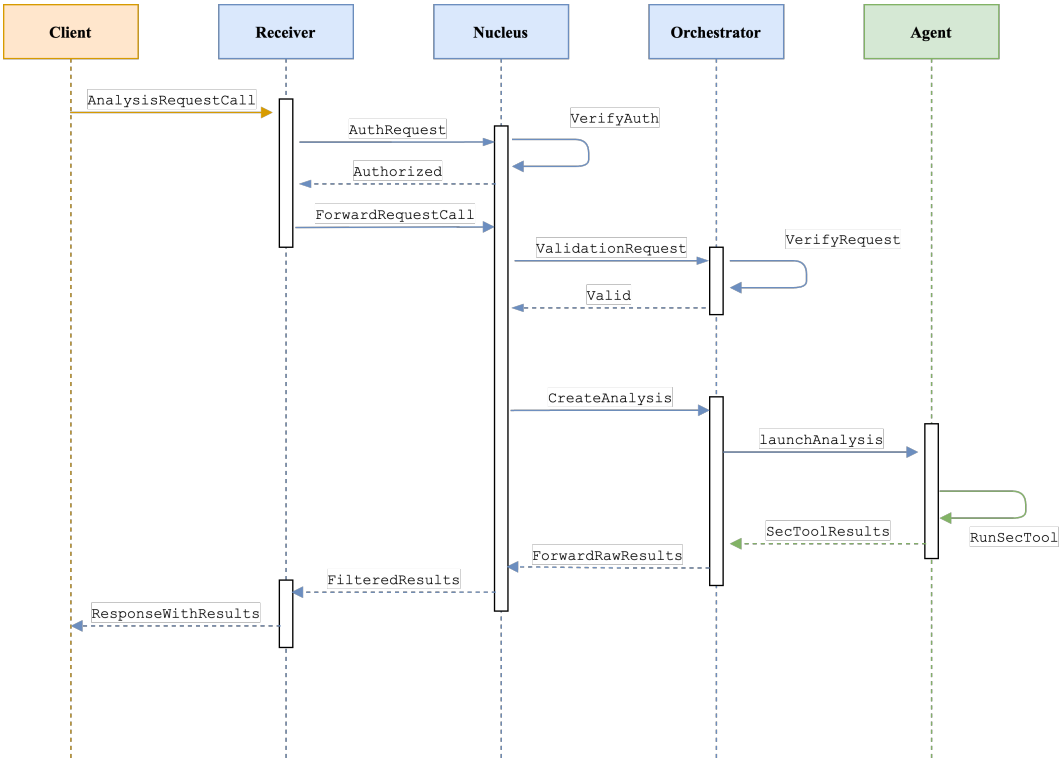


Figure 4.7: Message sequence chart - valid analysis request

Integration

The SecureApps@CI will be introduced and integrated into CI/CD system by calling the SecureApps@CI API on every pipeline that requires it, with the decision to be requested by software developers before an application pipeline is triggered. In terms of the procedure and process that SecureApps@CI supports, this solution has stages in which it performs different jobs for each one. The number and order of stages can be customized according to specific needs. In each stage, SecureApps@CI can run related jobs, and these can execute security tools. Each job will run inside a container working as an agent node triggered by the SecureApps@CI orchestrator. Every stage can be executed, skipped, or allowed to fail. We can define if jobs will run in series or in parallel on each stage so that the security tools are spin-up according to the inter-dependencies between stages. We can have stages that only run if a specific stage (e.g., secret detection) has a successful output.

CI/CD pipeline integration

A CI/CD pipeline is a set of steps to be followed to deliver a new version of an application. An API call to the SecureApps@CI API is required to request a new security analysis inside a CI/CD pipeline. This API call will upload a security analysis document to the SecureApps@CI solution. The security analysis document file will have a predefined format, and only that format is allowed. Then, the CI/CD pipeline starts, the security analysis job or task is reached, and a call to SecureApps@CI API is executed. Afterward, the security analysis runs, and

finally, the SecureApps@CI API returns a response code.

Agent nodes' jobs

The agent nodes receive jobs from the SecureApps@CI orchestrator. The jobs that an agent node runs may use local services or remote depending on the security analysis document. Local services are characterized by a restricted execution environment, based on open-source applications and without internet access (offline). Remote services are provided by external entities, with the inherent risk of publicly exposing sensitive data, without contracts or subscriptions involved and the absence of inherent guarantees.

Analysis procedure

The security analysis settings are loaded from a document (file) that acts as a form definition. The document containing analysis settings must exist inside the application code repository. In a typical application development pipeline, tracking changes and trigger pipelines are based on events in the Version Control System (VCS). A push or pull request on a repository of code will trigger the application's pipeline to run. We follow the same principles in our security analysis procedure. Therefore, a change in the form definition, followed by a push or pull, will trigger a new cycle of the application pipeline and, consequently, a call to SecureApps@CI API that runs a new security analysis.

The solution is a security analysis that provides a self-service analysis service inside CI/CD pipelines. Pipeline-as-code[4] defines a series of capabilities that allow users to establish mapping job processes with code housed and versioned throughout the source code repository. To achieve better results, avoid errors, roll-back configuration, and keep consistency, the application code has to be under source control. [5]. To have a clear, straightforward, and working security analysis is mandatory to have the security analysis document inside the application repository. The document inside the code repository enables organizations to track changes and audit the timeline history of security analysis settings.

Analysis result

The security analysis result includes a Boolean output response and a set of detailed reports. The Boolean output response is equivalent to a true or false, like in a common CI/CD pipeline of an application will have a `failed` or `success` status [6], here we have `passed` or `failed` in the security analysis pipeline.

[4] https://docs.cloudbees.com/docs/admin-resources/latest/pipelines/pipeline-as-code
[5] https://aws.amazon.com/devops/source-control/
[6] `https://www.jenkins.io/doc/book/pipeline/syntax/#post-conditions`

4.5.1 Security analysis flow

In Figure 4.8 we illustrate the security analysis flow, including authorization, verification, orchestration, and describe the respective requests and possible responses from a high-level perspective. The analysis flow shows a client (developer or CI/CD pipeline) sending a request to the receiver interface. This request can be authorized or not depending on the key provided is valid or invalid. The received request passes a verification to check if it is well-formed (OK). The orchestration parses, filters, and launches the security tools in series or parallel according to the security analysis definition form. The responses from security tools will go to a verification that will process and translate the information received to a proper HTTP status code, providing feedback to the client that originated the request. The status codes available are success codes (2xx), [7], unauthorized code (403), [8] and client error code (4xx) or server error (5xx). [9]

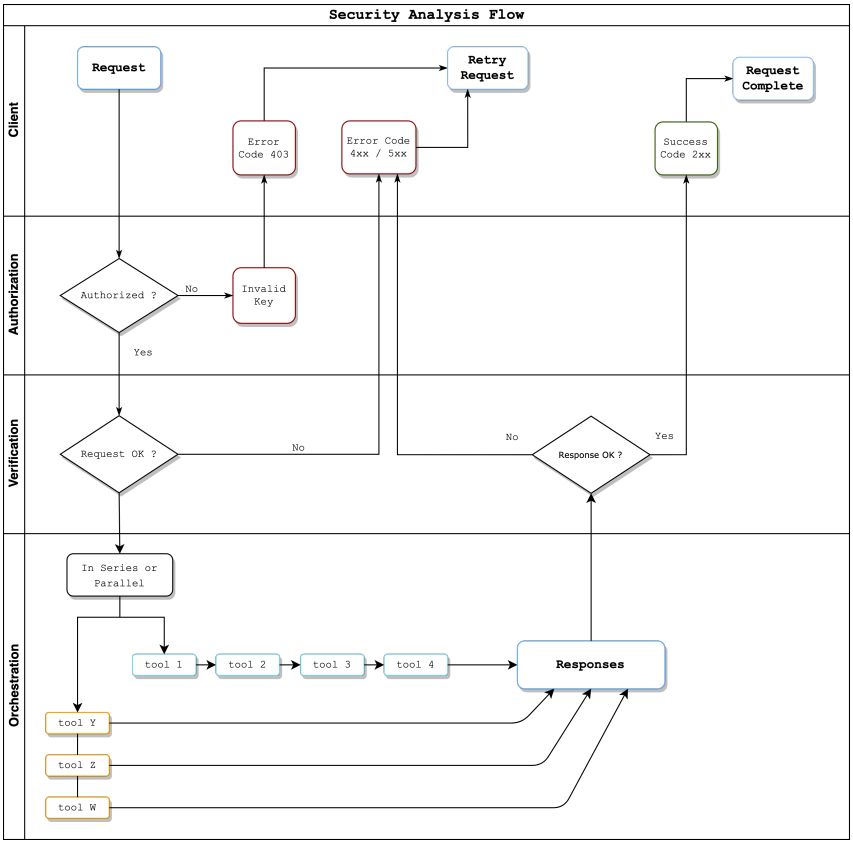


Figure 4.8: Security analysis - API flowchart

[7] https://httpstatuses.com/200
[8] https://httpstatuses.com/403
[9] https://httpstatuses.com/

CHAPTER 5

Implementation

This chapter will explain the available choices, decisions made, and how we implement the software components for a SecureApps@CI PoC. First, we explain the options available and why we chose a specific type of integration and a file format for security analysis, and a particular platform for orchestration. Second, we will do the analysis specification to determine the variables and parameters needed to define a security analysis in a consistent way. After determining the security analysis file format, the definition must describe the decisions to apply when specific conditions are met, establish prerequisites, or set precedences between stages. Third, we proceed with the API specification. The reason behind writing an API definition and what it provides.

5.1 Security solution options

For the SecureApps@CI integration, we have two options on the table to analyze: an API and a plugin. For the security analysis file format, the choices are between a standard Jenkinsfile[1] and YAML. For the orchestration platform, we need more than a container orchestration since there is a need to define and manage security analysis pipelines in a similar way that CI/CD pipelines for applications work. In CI/CD pipelines, the continuous integration and continuous delivery or deployment enforce the automation of the building, testing, and deployment phases of applications. In the security analysis pipeline, we need to automate the arbitrary security analysis regarding applications. More specifically, these pipelines will build the security tools involved, run correspondent security tests, and retrieve their results.

We intend to have a security analysis described as code using a descriptive model, with versioning history as developers use for source code. Security analysis as code concept is similar to others that define infrastructure as code or pipelines as code. With this approach, the purpose is to bring stability, consistency, tracking changes, adaptability to faster development,

[1] https://www.jenkins.io/doc/book/pipeline/jenkinsfile

and transparency to developers using the security analysis pipeline as code. If a new change on the security analysis definition file is required, any team can create Pull Request (PR) on the repository. We can go even further if there is a proposal to change the definition of security analysis. In this case, an affected team element will create a pull request in an application repository. The PR needs to be accepted by those managing the repository; the various teams involved can discuss and agree on changes bringing value and transparency to organizations and their businesses.

Analysis integration

There were two alternatives analyzed, more concretely an API and plugin. We wanted to enable organizations to adopt the CI/CD system that suits them best, migrate from one CI/CD system to another, or even use more than one. A plugin must follow the development of the CI/CD systems, and every time a new version of the system is released, the security tool will need testing, adjustment, and integration to create a new plugin release. Finally, the plugin option is bound to a specific CI/CD system, to have a plugin for several systems, the development time increase further. An API it is platform agnostic, with well-defined and manageable endpoints. This means a faster and iteratively integrated with application development. Due to these reasons, and to warrant consistency and compatibility across CI/CD systems, we chose an API and discarded the plugin option.

Analysis file format

Jenkins[2] is a pillar, one of the ancient and widely adopted CI/CD systems. Jenkins project started in 2004, originally called Hudson created by Kohsuke Kawaguchi, working at Sun Microsystems. It uses a Jenkinsfile[3] with declarative syntax. However, we chose YAML instead of Jenkinsfile because of several factors. First, it is easy to read and write by humans and simple to parse by machines. Second, a declarative pipeline in YAML is broadly embraced by several CI/CD systems, such as GitLab CI[4], Go-CD[5], Azure DevOps pipelines[6] and Google Cloud Build[7], Jenkins X[8], among others. Third, software developer are already used to read and write in YAML. Finally, our solution uses a CI/CD system for orchestrating the security analysis pipeline, and this system uses YAML as the unique file format for defining CI/CD pipelines. Due to these factors, our choice of YAML is clear and straightforward.

[2]https://www.cloudbees.com/jenkins/what-is-jenkins
[3]https://www.jenkins.io/doc/book/pipeline/jenkinsfile/
[4]https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/
[5]https://www.gocd.org/
[6]https://azure.microsoft.com/en-us/services/devops/pipelines/
[7]https://cloud.google.com/cloud-build
[8]https://jenkins-x.io/

Container engine

The container engine chosen is Docker engine. Docker engine is a strict requirement given the context of the SecureApps@CI. The environment of organizations in the context of our analysis and the correspondent CI/CD pipeline deployments have Docker as the container engine. Additionally, Kubernetes is configured with Docker as the container runtime, and its Kubernetes that will run applications sanitized by our SecureApps@CI. Our CI/CD system uses Docker engine as the container engine that will spin-up the security tools involved in the security analysis pipelines.

Receiver and nucleus

The receiver and nucleus blocks of the architecture referred in figure 4.4 will run inside the same container to reduce latency and improve performance. The receiver exposes an API and receives requests from clients, while the nucleus verifies, processes, and manages the requests without any API exposure. Both blocks were developed in Python.

Orchestrator and visualizer

The orchestrator and visualizer are two blocks with distinct functionalities, but they will live inside the same entity in our case. The main reason to use a CI/CD system to assume these blocks' roles is that these systems include web terminals for showing running jobs, a binary output of pipelines, and in some cases interactive terminals. This feature can tell us if pipeline was canceled, suspended, failed, passed, and real-time log.

We analyzed two CI/CD systems, Jenkins and GitLab CI/CD. The Jenkins architecture is fundamentally Server (Master) plus Agents and could use Docker within pipelines. It is possible to have multiple containers with distinct images for different stages[9]. GitLab CI/CD architecture essentially has at least one GitLab instance (Server) and one or more GitLab Runners. Both Jenkins and GitLab have the possibility of having agents running in the Docker engine. Jenkins has the `jenkins/agent` image and GitLab the `gitlab/gitlab-runner` image. With both GitLab API and Jenkins API, it is possible to trigger a new pipeline cycle. Jenkins needs plugins to declare pipelines and environment configuration in YAML. There are experimental plugins and an incubated plugin to declare[10]. Since these plugins are not native to Jenkins, they could bring unstable integration and interactions to our security analysis pipeline. GitLab CI/CD requires less configuration for pipelines than other similar setups in Jenkins.

GitLab ships to its container registry, which is ready for CI/CD container workflow without installing, configuring, or maintaining additional plugins. GitLab Runners create and upload job artifacts to GitLab and afterward, they are downloadable as an archive to GitLab UI or GitLab API. These Job artifacts are a list of folders and files generated when a job finishes. This feature is crucial once a security analysis pipeline finishes or breaks, since the teams involved

[9] https://www.jenkins.io/doc/book/pipeline/docker/
[10] https://plugins.jenkins.io/pipeline-as-yaml/

may need to observe and analyze the job artifacts. The *Auto DevOps* functionality brings DevOps best practices to the project by automatically configuring the software development lifecycle. It can detect, build, test, deploy, and monitor applications. GitLab has protected runners, that allow protecting sensitive information, such as the deployment of credentials and secret detection, by allowing only jobs running on protected branches to access them. A full comprehensive comparison table is available here[11].

In terms of data persistence, GitLab uses Gitaly[12] to keep the Git repositories, PostgreSQL[13] to persist the GitLab database data, and Redis[14] to maintain GitLab job data.

GitLab is flexible in terms of job artifacts, it has a container registry built-in, consistency and diversity of runners, the pipelines defined in YAML are clean, expressive and straightforward, and finally, the *Auto DevOps* capability enables the possibility of triggering a new pipeline cycle with a change in `.gitlab-ci.yml` file plus a push to the repository. Our final choice was GitLab CI/CD due to the number of available options and functions that will provide to our final PoC implementation.

Agents and executors

The agent nodes are devices that run jobs or tasks defined in each state of the security analysis pipelines. In our PoC we used three different Hardware devices (Laptop, VM and Raspberry Pi) with two different architectures, AMD64 and ARM. Agents run the code specified in the YAML definition file of each security analysis pipeline. The SecureApps@CI implementation uses Docker as the container runtime plus agent nodes for launching the security analysis pipelines, and not container orchestration such as Kubernetes or Docker Swarm worker nodes. Worker nodes are used by container orchestration platforms whose sole purpose is to execute containers and do not participate in the distributed state or make scheduling decisions.

A GitLab Runner is an open-source application written in Go. There are three ways to install it, more precisely in a container, downloading a binary manually or using `rpm/deb` packages from a repository. It is officially supported on the Linux, Windows, macOS and FreeBSD operating systems and on x86, AMD64, ARM64, ARM and s390x architectures. In GitLab, there are three types of runners, based on the type of access required. Shared runners are available for all groups and projects inside a GitLab instance. Group runners are only available to projects and subgroups in a group, and specific runners are associated with specific projects.

When we register a runner, we must choose an executor. An executor determines the environment each job runs in. Different executors are available and can be chosen, like SSH, shell, VirtualBox, Parallels, Docker, and Kubernetes. We evaluated two alternatives for executors: Docker executor and Kubernetes executor. Our CI/CD system implements several

[11] https://about.gitlab.com/devops-tools/jenkins-vs-gitlab/decision-kit/
[12] https://docs.gitlab.com/charts/charts/gitlab/gitaly/index.html
[13] https://github.com/bitnami/charts/tree/master/bitnami/postgresql
[14] https://github.com/bitnami/charts/tree/master/bitnami/redis

executors that we can choose to run builds in different scenarios. It supports distinct platforms and methodologies for building a project.

The Kubernetes executor has all features enabled according to the compatibility chart of our CI/CD system[15], but does not have an interactive web terminal for the gitlab-runner helm chart. It allows the usage of an existing Kubernetes cluster for the builds. The executor calls the Kubernetes cluster API and creates a new Pod[16] to build and service containers for each GitLab CI job. Pods are small deployable objects in Kubernetes representing a single instance of a running process in your cluster. They can contain one or more containers, such as Docker containers. If a Pod runs multiple containers, Kubernetes manage them as a single entity and share the Pod's resources. The biggest drawback is that teams and organizations will need a functional Kubernetes cluster to perform safety analysis pipelines.

The Docker executor allows a clean build environment, with dependency management (all dependencies for building a project are inside a unique Docker image), and has all our required features enabled, including the interactive web terminal, unlike Kubernetes. It has the advantage that it is exceptionally straightforward to have Docker engine running on a developer laptop or agent nodes, as opposed to Kubernetes. We want to ensure maximum flexibility and independence from additional layers to perform our PoC. After reviewing the pros and cons of these alternatives, our choice was the Docker executor for our PoC.

Analysis results

The SecureApps@CI results of security analysis pipelines will be available through the API or a web interface. A Boolean output and a detailed report will be visible to both. For the presentation of the results, SecureApps@CI allows defining the amount and type of errors and saves or hides errors. There is a sort of cache for old errors detected that gives the possibility of showing only new errors (hiding previous ones), hiding errors equal to the earlier errors found, showing a small warning to count the number of prior errors hidden and view hidden errors while keeping context. The information regarding the errors must have a simple and easy-to-read format, preferably an output format for both machines or services and humans, like JSON. Other options can be a table or text output, in case JSON is unavailable. The components of the message to display are 3: precise location of the error in the code (folder, file, line, commit hash), a short description of the error or vulnerability, and a URL with information regarding the vulnerability, if any available.

5.2 Analysis specification

The definition of the security analysis' parameters is made in a YAML file[17]. The YAML definition can describe the decisions to apply when specific conditions are met. It can establish

[15] https://docs.gitlab.com/runner/executors/
[16] https://kubernetes.io/docs/concepts/workloads/pods/
[17] Security-Analysis.gitlab-ci.yml

prerequisites or requirements and set precedences in stages. For example, when a stage succeeds or fails if the security analysis will move to the next stage. It can also specify actions to be executed before or after a specific stage. For example, install dependencies before evaluating an application or sending a notification via email/slack after. Parameters, conditions, and policies can be set, defined as setting different environments, Quality Assurance (QA), development, staging, or production, and defining types of checks like detection of secrets, static or dynamic analysis. We can also define the structure, what to execute, the order of stages, or jobs. There are other possibilities, like skipping stages and white-list checks according to organization security policies (for example, setting a vulnerability has accepted in the development environment). The parameter setting of the SecureApps@CI was based on the "GitLab CI pipeline configuration reference"[18] (Community Edition).

```yaml
## Global variables. These variables can be accessed in any job.
variables:
  ENV: "qa"

## stages definition
stages:
  - stage1
  - stage2

## jobs definition
job1:
  variables:
    JOB_NAME: static_analysis
  stage: stage1
  script:
    - make $JOB_NAME
  only:
    - qa
  allow_failure: false

job2:
  stage: stage2
  script:
    - make clean
  allow_failure: true
```

Code Snippet 1: gitlab-ci.yml sample

With sample configuration in Code Snippet 1, we will briefly introduce the GitLab CI configuration reference. There are three key components, variables, stages and jobs. Variables can be defined as global in the top of YAML definition or attached to specific jobs. Stages determine the phases or steps of a CI/CD pipeline and its ordinance determines the sequence of execution. Jobs sequentially in the order specified in stages. In this case 1, `job1` from `stage1` will run first, and `job2` of `stage2` will run after. All jobs inside the same stage run in parallel. It is also possible to establish dependencies between jobs with `needs` keyword and execute jobs in multiple stages concurrently.

[18] https://docs.gitlab.com/ce/ci/yaml/README.html

Next, in Code Snippet 2, we present a security analysis template sample including two stages (`secret_detection`, `static_analysis`). The `secrets` job is part of `secret_detection` stage will run first, and afterward the job `sast` from `static_analysis` stage will be executed. Job `secrets` can only run on dev and staging environments, which produce artifacts and do not allow failure. The `sast` job depends on `secrets` job and will be executed only if `secrets` job passes on success. This job allows failures, meaning that if the job fails, the pipeline continues to the next stages and correspondent jobs if they exist. The analysis can also have a `.pre` or `.post` stage that will run before and after all stages, respectively. The stages can be added or removed according to the analysis requirements with the following example of a configuration.

```
# Global variables. These variables can be accessed in any job.
variables:
  APP_NAME: "app2"
  # name of the "unit/division/section/department/team/workgroup/work_project"
  GROUP_NAME: "qa-team"
  GIT_URL: "git@<git server URL>:<port>/<project/user key>/<repository slug>.git"
  BRANCH_NAME: "feature-yz"
  GIT_DEPTH: '1'
  ENV: "qa"

stages:
  - secret_detection
  - static_analysis

secrets:
  stage: secret_detection
  script:
    - TESTS="detection" make run    # run secret detection tool(s)
  only:
    - dev
    - staging
  artifacts:
    paths: ["${SECRETS_RESULTS}"]
    when: always
  allow_failure: false

sast:
  stage: static_analysis
  needs: ["secrets"]
  script:
    - TESTS="static" make run       # run static app-sec tool(s)
  only:
    - dev
    - staging
  artifacts:
    paths: ["${SAST_RESULTS}"]
    when: always
  allow_failure: true
```

Code Snippet 2: Security Analysis template

A generic CI definition file called `.gitlab-ci.yml` was also added to the analysis repository by our SecureApps@CI PoC. The generic CI definition shown in Code Snippet 3 is a pipeline as code file with a key including the security analysis and variables related to each analysis.

We can use these variables to identify the analysis later on, like the analysis id, path, project, repository, and workgroup. Notice that the project is the slug of the source code repository. The analysis id is a base64 encoded string generated from a 48-byte random stream. The existence of a file called `.gitlab-ci.yml` is mandatory since the *Auto DevOps* capability is only possible with it. If this file changes, it will trigger new security analysis pipelines achievable using GitLab server and consequently running jobs on correspondent runners.

```
include:
- local: Security-Analysis.gitlab-ci.yml
variables:
  # base64 encoded string generated from a random 48-bytes long stream
  ANALYSIS_ID: Sample+Value/zJhwYgiwtySWJX+zO3Yo9YIFuv4/d56TcS6ZeXGBL0q3CC1lYL5
  ANALYSIS_PATH: ./_analysis/app2
  ANALYSIS_PROJECT: qa-team-app2
  ANALYSIS_REPO: git@lab.example.com:analysis/qa-team-app2.git
  ANALYSIS_WORKGROUP: analysis
```

Code Snippet 3: Generic GitLab-CI definition

5.3 Solution orchestration

This section will approach the details and specifics of the SecureApps@CI API, more concretely its orchestration API. Our goal is to analyze containerized applications in isolated environments. On each CI/CD pipeline for applications where there is a security definition file in YAML format, it will perform a security analysis pipeline. For each job or task (secret detection, SAST, DAST or others) of the security analysis, there is a Docker image of a security tool that will run inside a container.

The orchestration is based on an orchestrator and agents, namely, a GitLab server and runners. It is a push model, so the runners send back the results to the GitLab server. To orchestrate the SecureApps@CI solution, several pieces need to interconnect and communicate with each other. All components will run inside containers. The main components are the API server, backend, GitLab-CI, runners and container engine. In our context, GitLab-CI will work as orchestrator and visualizer. Orchestrators[19] are tools to manage, scale, and maintain containerized applications. A container engine[20] is a piece of software that accepts user requests, including command-line options, pulls images, and from the end user's perspective, runs the container.

[19] https://docs.docker.com/get-started/orchestration/
[20] https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction

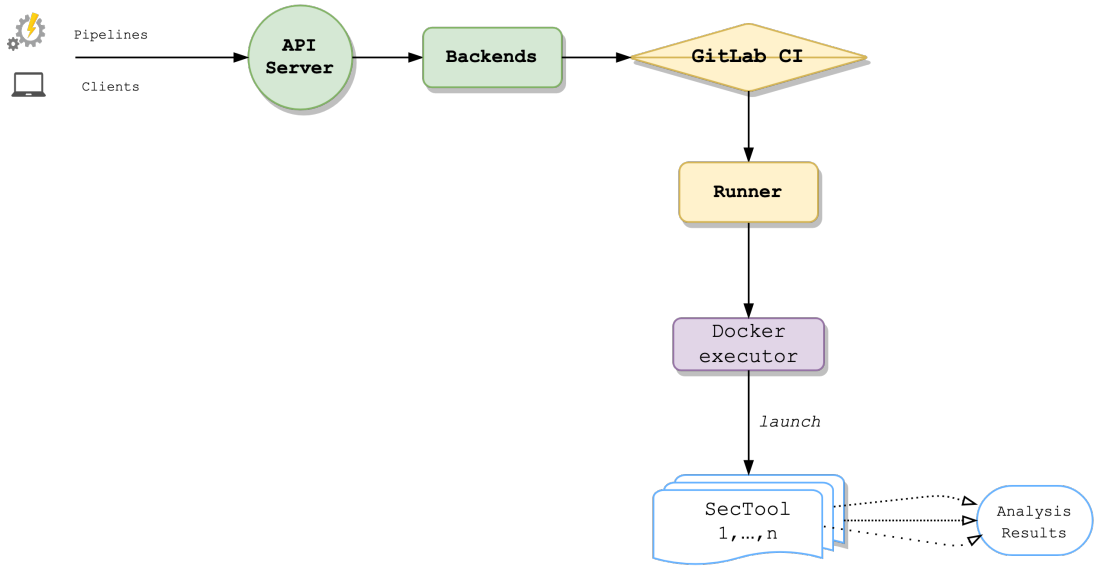


Figure 5.1: API orchestration

As shown in figure 5.1 the API server receives requests from clients or CI/CD pipeline, sends these requests to API backend that filter, lint and verify its contents. After this, the GitLab-CI sends requests (with actions) to its runners. Runners communicate with Docker executor that launches security analysis jobs. These jobs include security tools that run inside the agents, in our case Docker containers. These tools can also run on other types of agents, like Amazon EC2 instances, Google Compute instances, or even VMs can be applied if SecureApps@CI is adapted accordingly.

All security tools will run inside a security analysis pipeline, produce results, and send them back to clients (developer) or make it available within CI/CD pipeline. API server and backend have the same color since they will work as the same entity and run on the same container. The GitLab and Runners also have the same color because they are part of the same entity.

For flow and execution conditions, there are: jobs performed if a failure exists, some applicable in specific environments and others never executed in particular contexts. Next, we present examples of different types of conditions:

- `when:on_failure`, defines a job performed if a failure exists
- `only:staging` defines a job only applied in staging environments, for example.
- `except:master` defines job that are never executed in the master branch
- `needs:secret-detection` defines a job that only runs if job secret detection finishes successfully
- `artifacts:true` defines a list of files and directories to attach to a job on success.

For notification, the SecureApps@CI solution will have distinct return status 0, 1, and 2; 0 for `success`, 1 for `fail`, and 2 for errors that CI/CD systems receive and afterward return errors and send alerts.

5.4 CI/CD PIPELINE VERSUS SECURITY ANALYSIS PIPELINE

A security analysis pipeline is distinct from a CI/CD pipeline of an application, even if it can run on the same system. Figure 5.2 shows how a security analysis pipeline analysis fits into a CI/CD pipeline of an application that is intended to be sanitized and cleaned. In a typical CI/CD pipeline with three stages, build, test, and deploy, there will be a job that calls the SecureApps@CI API during the test stage. Upon this call, a new security analysis is triggered and a distinct security tool will run tests A, B, and C. These tests will run also in a pipeline on a CI/CD system called GitLab CI, so this means we have a security analysis pipeline inside a CI/CD pipeline.

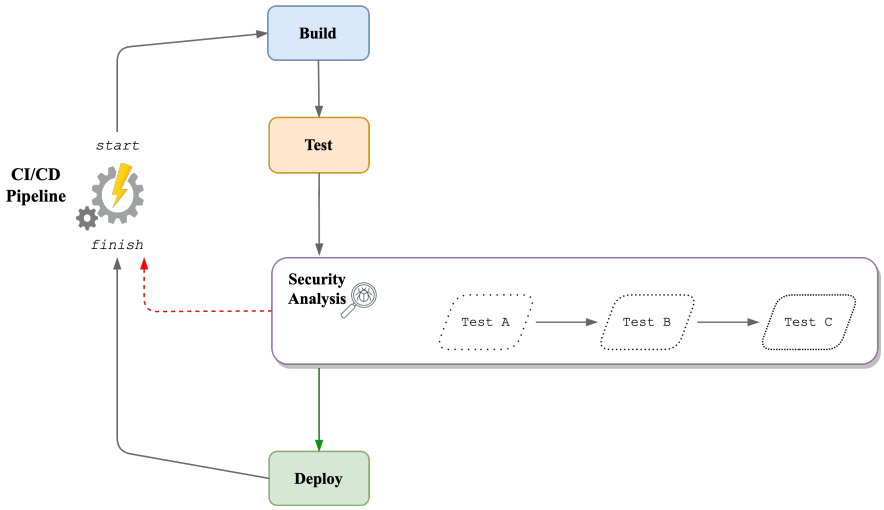


Figure 5.2: Security analysis pipeline in a CI/CD application pipeline

Our purpose is to trigger a new security analysis pipeline cycle with a source code change on the master or specific branches if needed. GitLab CI triggers[21] can be used to force a pipeline rerun of a particular reference (branch or tag) with an API call. Although triggering pipelines with GitLab API is possible, we follow a typical CI/CD pipeline's principles. When code changes are pushed to a repository, a new application pipeline starts. In our PoC, after *Auto DevOps* is enabled, new security analysis pipelines will run every time the source code changes in the Git repository of applications in the desired branch. This behavior is similar to a typical CI/CD pipeline of an application that will also react to code changes and trigger new pipeline cycles. *Auto DevOps* is the default CI/CD model, allowing GitLab to automatically detect, compile, test, deploy and monitor applications. *Auto DevOps*[22] intends to simplify the setup and execution of a mature and modern SDLC.

[21] https://docs.gitlab.com/ce/ci/triggers/README.html
[22] https://docs.gitlab.com/ce/topics/autodevops/

5.5 API SPECIFICATION

The reason behind writing an API Definition[23] is that an OpenAPI Specification (OAS) provides a contract that describes what responses look like when someone calls the API. This definition can help the internal developers understand the API, agree on its attributes, and help external developers learn the API and what it can do. In addition to generating documentation, OAS accelerates development by creating implementation code and SDKs. Other advantages are creating tests, monitoring[24], and bringing the API live. With tools like API Contract Security Audit[25], it is feasible to get a detailed analysis of the possible vulnerabilities and other API contract issues by providing the API contract in JSON format. Developers can write different kinds of tests like functional tests, load tests, and run security scans with ReadyAPI[26].

5.5.1 API DEFINITION

For a functional API definition, it is necessary to declare items like API server and base URL, paths, operations, parameters, requests, responses, data models (schemas), authentication methods, and optional examples. In OpenAPI[27] terms, paths are endpoints (resources), such as `/users` or `/reports/summary`, that the API exposes, and operations are the HTTP methods used to manipulate these paths, such as GET, POST, or DELETE. The RESTful[28] API was developed with Swagger[29], a tool ecosystem for developing APIs with OAS. The final SecureApps@CI API is available online at SwaggerHub[30]. During its development, we used API auto-mocking integration[31]. This integration creates and maintains a mock of APIs based on the responses and examples described in OAS. API Auto Mocking enables development, iteration, and testing requests and responses against the mock API backend. This helps during the design phase and building client applications even before the real API backend is ready.

Just as an example, a simple CLI tool called `curl`[32] can call the health status method on API Auto Mocking[33] endpoint, like the following example:

[23] https://swagger.io/blog/api-development/why-you-should-create-an-api-definition/
[24] https://support.smartbear.com/alertsite/docs/monitors/api/soapui/create.html
[25] https://apisecurity.io/tools/audit/
[26] https://support.smartbear.com/readyapi/docs/projects/create/swagger.html
[27] http://spec.openapis.org/oas/v3.0.3
[28] https://www.redhat.com/en/topics/api/what-is-a-rest-api
[29] https://swagger.io/tools/open-source/getting-started/
[30] https://app.swaggerhub.com/apis/ema.rainho/secureapps-ci/v1
[31] https://app.swaggerhub.com/help/integrations/apiautomocking
[32] https://curl.se
[33] https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/

```
$ curl -X GET https://virtserver.swaggerhub.com/ema.rainho/secureapps-ci/v1/health
{
  "api_status" : "up"
}
```

Code Snippet 4: Sample API call with curl

5.5.2 API METHODS

There are three groups of methods 5.3 for calling the SecureApps@CI solution: `generic`, `tools`, and `analysis`. The `generic` methods are `health` and `version`. The `tools` methods are: `create`, `list` and `details`. Finally, the `analysis` methods are: `create`, `abort`, `progress`, `jobs` and `results`.

All API methods are described in detail in section 7.1. The generic methods of `health` and `version` are informative. Although they seem unnecessary and harmless, they can bring precious value. Knowing whether the API is available is essential for both clients and CI/CD pipelines. Knowing the exact version of the API helps to avoid mistakes or unexpected results. It helps to maintain consistency in the CI/CD pipelines that include our security analyses, keeping them as deterministic and effective as possible.

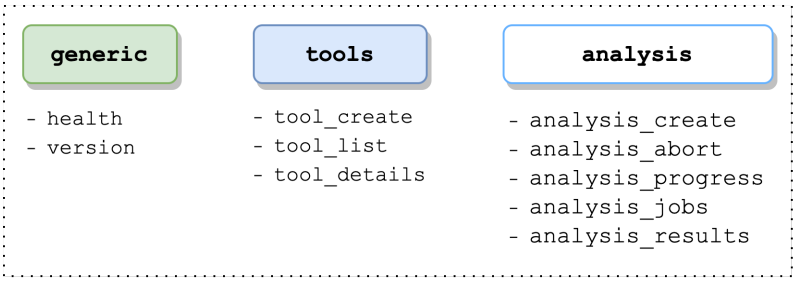


Figure 5.3: API methods

Before calling an API, the first approach is to check health status; this way clients ensure that it is available online and can reach it. After this verification, clients may start to do other requests and operations on the API. The health status of the API can be retrieved by a developer or a CI/CD pipeline using a GET HTTP method to query the API endpoint `/analysis/health` (see figure 5.4). The orchestrator will query the container engine, which in turn will check if the API node is running and has the path available. The API node returns an exit status to the container engine. If API node is `up`, returns an exit code of zero (0); if `down`, the exit code is one (1). Otherwise, in case of errors, the exit code is two (2). After receiving the output code, the orchestrator builds a JSON response with a key called `api_status` and values `up` or `down`. Notice that if the output of container engine is two (2), it means an error occurred, and the `api_status` value will be down.

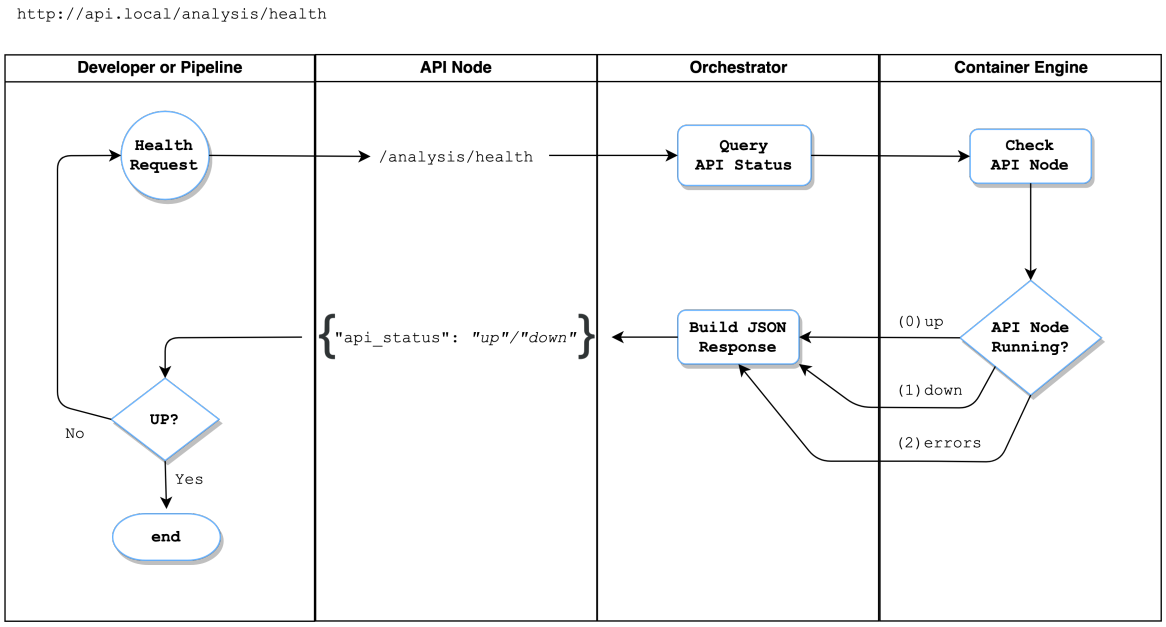


Figure 5.4: API - health method

After the first approach to check the API health status, clients need to verify the versions available to read the documentation, determine the desired methods and query the API accordingly. After this verification, clients may start to do other requests and operations on the API. The version of the API can be retrieved by a developer or a CI/CD pipeline using a GET HTTP method to query the API endpoint `/analysis/version` (see Figure 5.5). The orchestrator will query the container engine, which in turn will check the API node versions available. The API node returns an exit code of zero(0), and a version number if succeeded or returns an exit code of two(2) in case of error. After receiving the exit status, the orchestrator builds a JSON response with a key called `api_version` and a value `v1` for example. Notice that if the output of the container engine is two (2), it means an error occurred, and the `api_version` value will be an empty string.

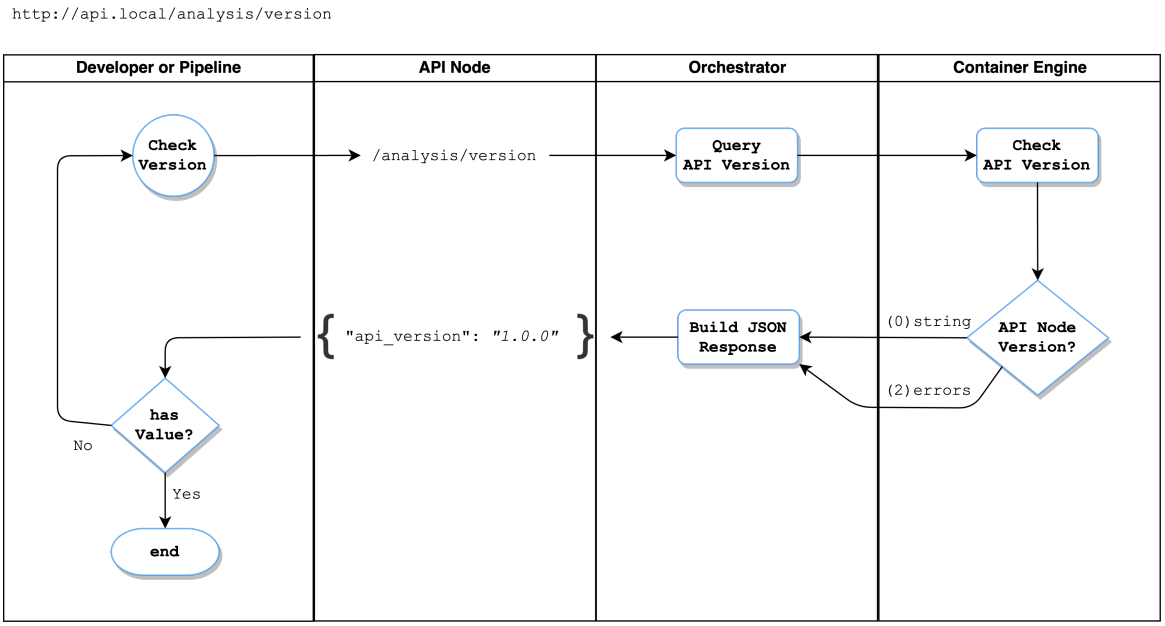


Figure 5.5: API - version method

We can create a golden image of security tools to ensure consistency across an analysis, avoid errors, and reduce the security analysis time. In SecureApps@CI the golden image is achieved by building a Docker image with a security tool installed and properly configured. With this approach, developer and CI/CD pipelines take less time to have security analysis results since the golden image of security tools has been built and referenced in the analysis definition form. Three helpful methods can improve security analysis quality and speed, although they are not mandatory. They are the `tool_create`, `tool_list` and `tool_details` methods. The creation of a new security tool, more specifically the `tool_create` (see Figure 5.6), can be requested by a developer or a CI/CD pipeline using a POST HTTP method to send a request to the API endpoint `/tools/create`. This request includes a Dockerfile that contains the settings to build the intended security tool. The orchestrator parses the Dockerfile and triggers a lint tool in the container engine to ensure the Dockerfile is valid. In case this Dockerfile is accepted, the exit code is zero(0), and one(1) otherwise. After receiving the exit status from container engine, the orchestrator builds a JSON response containing the keys `tool_id`, `tool_name` and `accepted`. If a new tool is created, this process ends; else, we can do a new request. In the creation of a new tool, the `tool_id` includes an `image-name` and a tag like `image-name:tag` (for example `nginx:alpine`) and `tool_name` will be the container name.

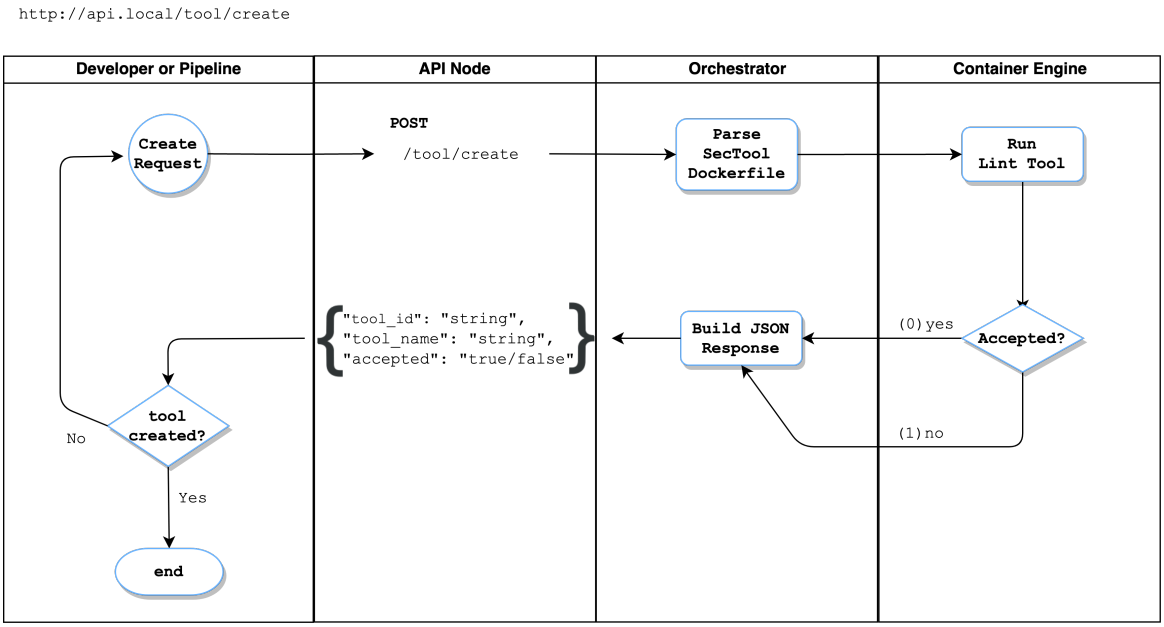


Figure 5.6: API - method to create a new analysis tool

Upon creating a security tool with the `tool_create` method, a developer may want or need to check and evaluate the tool definition's contents. This can be achieved by having access to the Dockerfile regarding the security tool to evaluate. To achieve this goal, we create a method called `tool_details` that will super seed this need. The details of a specific security tool can be retrieved by a developer or a CI/CD pipeline using a GET HTTP method to query the API endpoint `/tool/toolId/details` (see Figure 5.7). The orchestrator will query the container engine to find the security tool details and collects them. If any details are collected, the exit code is zero (0) and one (1) otherwise. After receiving the exit status from the container engine the orchestrator builds a JSON response containing the definition of security tool with keys `tool_id`, `tool_name`, `dockerfile_id` and `dockerfile_dump`. If the details are retrieved, this process ends, else a new request can be made.

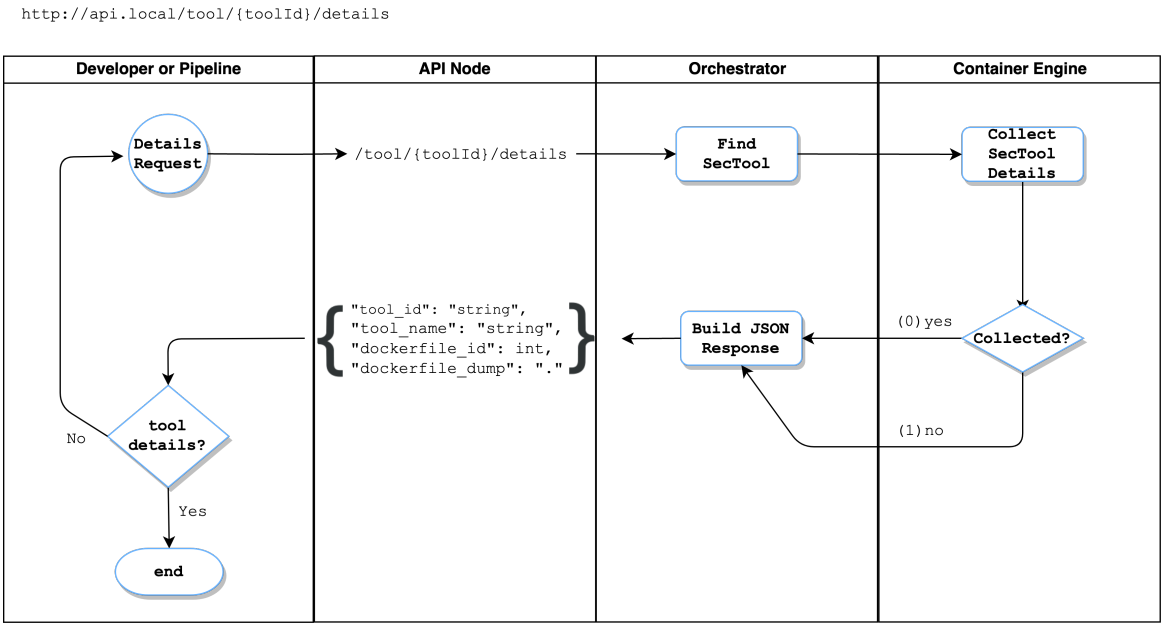


Figure 5.7: API - method to fetch the details of existing analysis tools

Since we already have a method to add a new security tool to our security analysis pipelines and another method to retrieve the definition of an existing tool, it makes sense to have a method to list all existing tools: `tools_list` (see Figure 5.8), where a developer or a CI/CD pipeline can request it by using a GET HTTP method to query the API endpoint `/tools/list`. The orchestrator will find available security tools after container engine collects their images. If any security tools are collected, the exit code is zero (0), and one (1) otherwise. After receiving the exit status from container engine, the orchestrator builds a JSON response containing a list of security tools with keys `["tool1": "xyz","tool2": "ijk", "tool3": "uvw", "..."]`. If the list is retrieved this process ends, else a new request can be done.

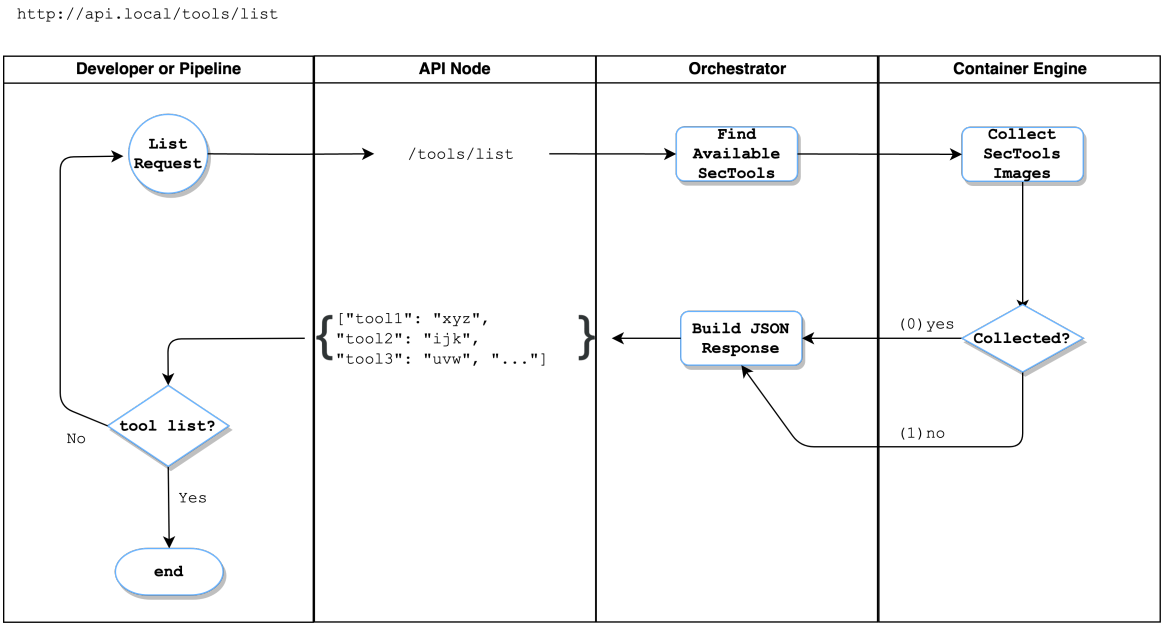


Figure 5.8: API - method to list all existing analysis tools

With the purpose of determining if applications have security issues and sanitizing them before they are released, we established specific methods to manage security analysis pipelines: create, abort, progress, jobs and results. We start by defining the `analysis_create` method that enables a developer or a CI/CD pipeline to create a new security analysis pipeline (see Figure 5.9). A YAML definition in a source code repository of a certain application (App1 or App2) is uploaded via a POST request to the API node. The orchestrator parses the YAML definition file. A container will run a lint tool returning zero (0) to accepted requests and one (1) otherwise. After this, the orchestrator builds a JSON response with keys `analysis_id`, `analysis_name` and `accepted`; `accepted` can have values true or false. If the analysis is accepted, the flow ends. If not, a new create request can be executed if needed.

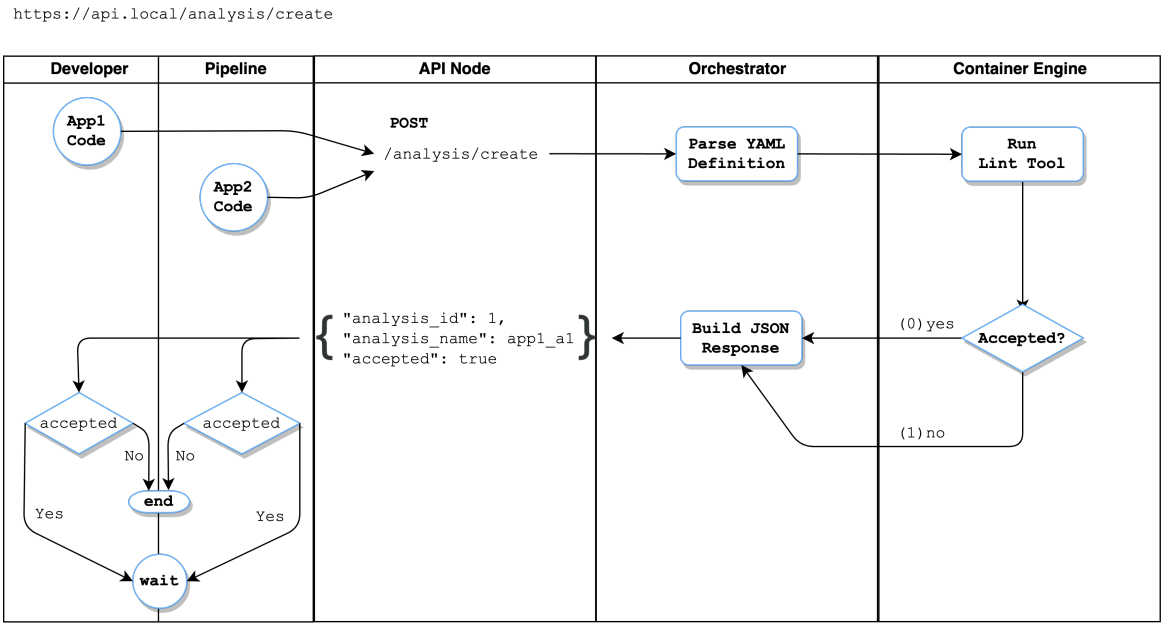


Figure 5.9: API - method to create a new security analysis pipeline

After creating a security analysis pipeline, the developer or a CI/CD pipeline may need to abort it. For instance, because they realized that there is an issue. Also, if an application's new feature or functionality is added to a branch, merged to master, if that application's CI/CD pipeline is still running, the team may want to cancel it, including the security analysis pipeline, before the new feature is added. These two cases show it is useful and valuable to abort a current analysis pipeline to reduce time, decrease costs, and avoid incorrect or inconsistent reports. To solve this need, we created a new method called `analysis_abort` that will abort a security analysis (see Figure 5.10). It can be requested by a developer or a CI/CD pipeline using a GET HTTP method to query the API node on the endpoint `/analysis/analysisId/abort`. The orchestrator will query the container engine to find security tools related to the analysis referenced by *analysisId*. The container engine stops the security tools of the analysis; if they stop correctly, the exit code is zero (0), one (1) if they did not stop, and two (2) in case of any errors. After receiving the exit status from the container engine, the orchestrator builds a JSON response containing keys `analysis_id`, `abort_status`. The `abort_status` can have two values, `success` or `fail`, if it is `success` this process ends, else a new request can be done.

```
https://api.local/analysis/{analysisId}/abort
```

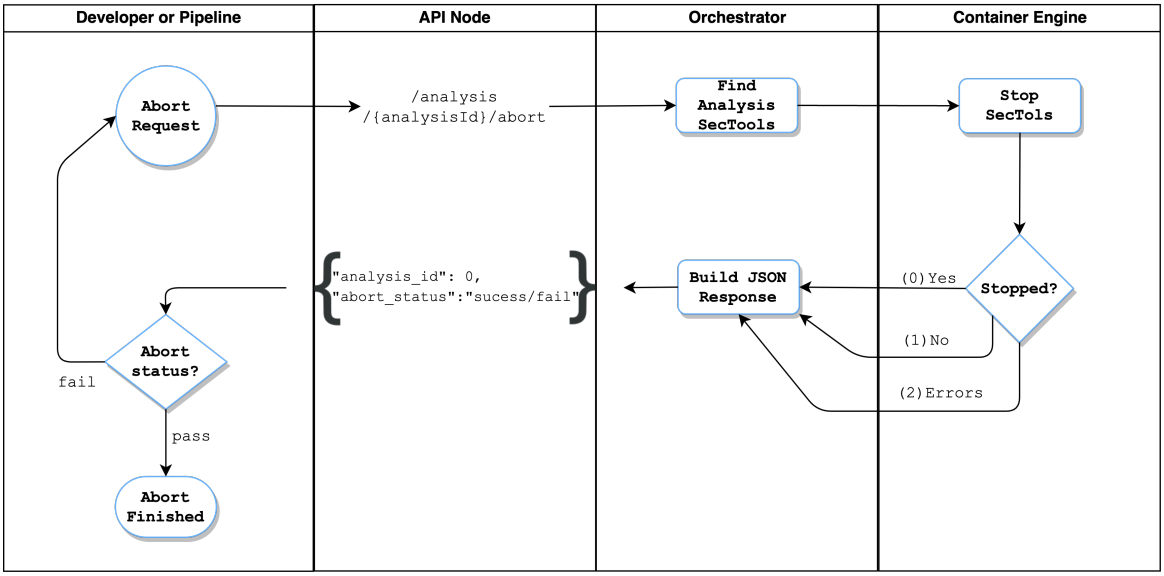


Figure 5.10: API - method to abort an existing security analysis pipeline

Developers or a CI/CD pipelines may need to verify the evolution of an analysis pipeline to determine its current state and how long it may take to finish. This is useful, when a developer calls the API for analyzing an application and the analysis is running longer than expected, for instance, on an otherwise short test, already performed in isolation. In these cases, it is important to consult the progress and status of the analysis pipeline. For this, we created a new method, called `analysis_progress` to monitor the progress of a security analysis (see Figure 5.11). Such progress can be retrieved by a developer or a CI/CD pipeline using a GET HTTP method to query the API endpoint `/analysis/progress`. The orchestrator will query the container engine to find security tools related with analysis referenced by `analysisId`. The container engine checks security tools progress related with the `analysisId`, if all of them are finished, the exit code is zero (0), one (1) if they did not finish and two (2) in case of any errors. After receiving the exit status from container engine, the orchestrator builds a JSON response containing keys `analysis_id` and `progress`. The `progress` can have three values `accepted`, `ongoing` or `completed`. If it is `completed` this process ends, if it is `accepted` or `ongoing` new requests are done until the analysis completes.

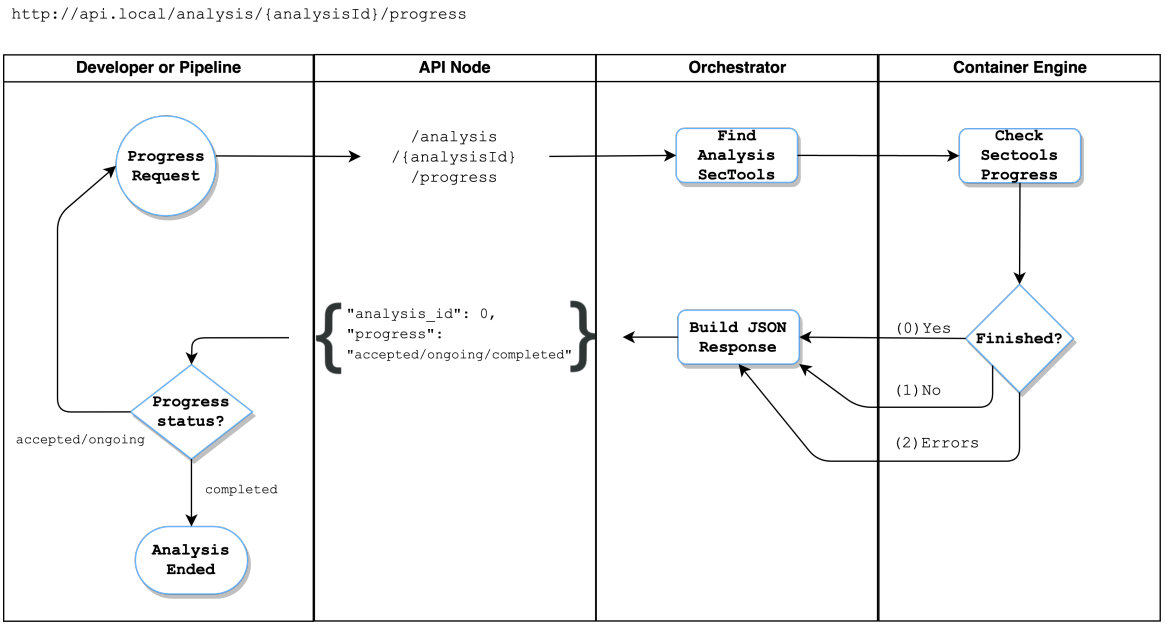


Figure 5.11: API - method to verify the progress of an analysis pipeline

In the sequence of method `analysis_progress`, that monitors the progress and evolution of a security analysis pipeline, developers or CI/CD pipelines may need to access the jobs belonging to each stage of that pipeline. Having this information is more comfortable and strait-forward to determine the current state and have a better view of how long it will take to finish the whole security analysis pipeline. It is beneficial, for example, if a team is testing a new application, without having any idea of a particular test of a security tool, and needs to check beforehand if that specific test has already run or not. The jobs for a stage in an analysis pipeline can be retrieved by a developer or a CI/CD pipeline using a GET HTTP method to query the API endpoint `/analysis/analysisId/stageId/jobs` (see Figure 5.12). The orchestrator will query the container engine to find security tools from stage `stageId` within an analysis `AnalysisId`. The container engine queries security tools' jobs related with `stageId`, if there are jobs, the exit code is zero (0), if no job is found the code is one (1) and in case of any errors, two (2). After receiving the exit status from the container engine, the orchestrator builds a JSON response containing keys `stage_id`, `stage_name` and `job_list`. If `jobs_list` is empty, it means the analysis pipeline already completed and all process ends, else a new request can be done until the analysis reaches the end.

https://api.local/analysis/{analysisId}/stage/{stageId}/jobs

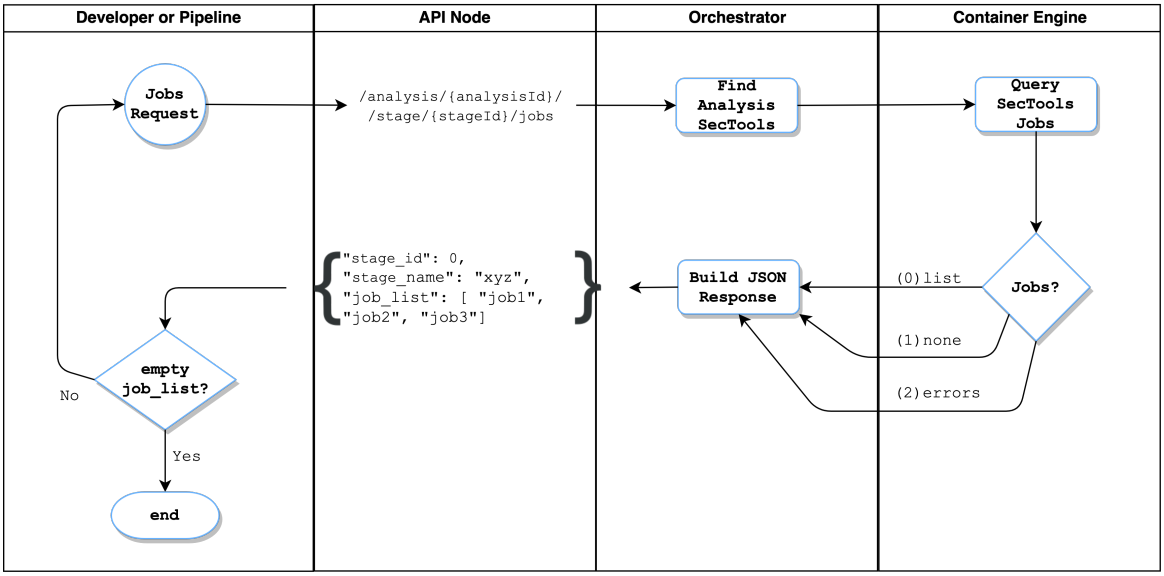


Figure 5.12: API - method to query the jobs running in a given stage of active analysis pipeline

The latest and crucial part of an security analysis pipeline is their result, namely, if it passed or failed. Furthermore, in the event of failure, have a detailed report of why it happened. To do this, we need to determine the type of security test that failed, visualize the reason for the failure, analyze the type of error and act accordingly. This method, `analysis_results`, is fundamental to SecureApps@CI solution since without it, the solution is useless, the developer and CI/CD pipeline had no feedback of results, and the solution did not fulfill its purpose. The results of an security analysis can be requested by a developer or a CI/CD pipeline as described in Figure 5.13. The orchestrator will query the container engine to find the security tools related with analysis referenced by `analysisId`. The container engine queries the security tools responses related with the `analysisId`. If there are no security issues the exit code is zero (0), and one (1) if one or more security tools found security issues and two (2) in case of any errors. After receiving the exit status from the container engine the orchestrator builds a JSON response containing keys `analysis_id`, `analysis_status`, `report_id` and `report_results`. If no security issues were found, the application code (App2 Code) will pass, and a pull request or push is accepted. If any security issue was found, App1 Code will not be pushed, and App2 Code on the CI/CD pipeline will fail.

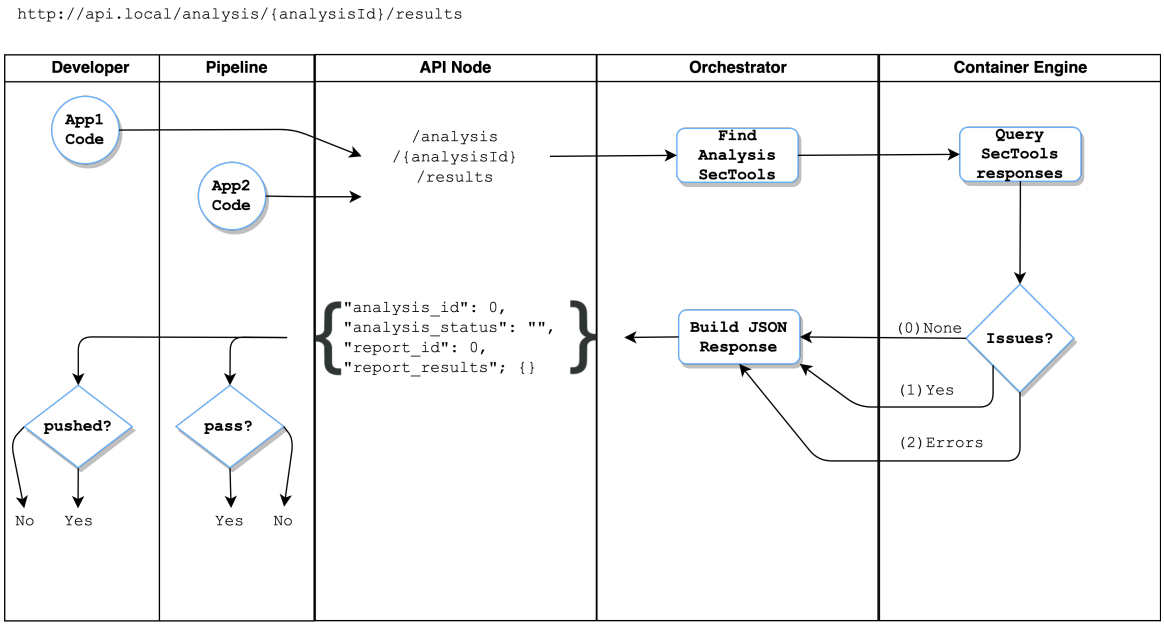


Figure 5.13: API - method to get the results security analysis pipeline

5.6 Software structure

Our complete system uses three distinct APIs, specifically by SecureApps@CI API, GitLab CI API and Docker engine API. The first was implemented according to the API specification in section 5.5. The other two APIs to tackle security analysis requests and fulfill its purpose. The software structure of our system is composed of various packages and classes that define the various blocks of architecture. The components implemented were the API server and backends. The two main packages involved were the `swagger_server` to the API server code and `api_backend` referent to the backends code. The API server receives requests from clients, backends process them and communicates with GitLab CI API and Docker engine APIs to provide the required actions to the security analysis pipeline. The API server definition contract is encapsulated in a `swagger.yaml` file, that contains all endpoints, methods, controllers, models, authorization, actions and responses related to the API server of our PoC. The written code for SecureApps@CI, namely the API server and backends, is Python and are available at GitLab[34]. For the GitLab CI API[35] interaction, we used the Python package `python-gitlab`[36]. For the Docker engine API[37] interaction, we use the Python SDK, more specifically a Python library `docker-py`[38]. The API server implementation uses `connexion`[39], that is, a Python framework built on top of Flask, which handles HTTP requests defined using OpenAPI (aka Swagger). In all cases, we used version 3 of the specification. We start by defining the classes, their attributes, and methods needed to fulfill the requirements of

[34]https://gitlab.com/secureapps-ci/api_server
[35]https://docs.gitlab.com/ce/api/README.html
[36]https://github.com/python-gitlab/python-gitlab
[37]https://docs.docker.com/engine/api/v1.40/
[38]https://docker-py.readthedocs.io/
[39]https://github.com/zalando/connexion

our system. From the classes diagram of Figure 5.14 we can observe the relations between the different classes involved. The main classes used are `APIActions`, `GitLabConnector`, `Analysis`, `Results`, `DockerActions` and `SecTool`. The `DockerActions` class is optional, dedicated to reducing errors, improving the consistency and increasing performance.

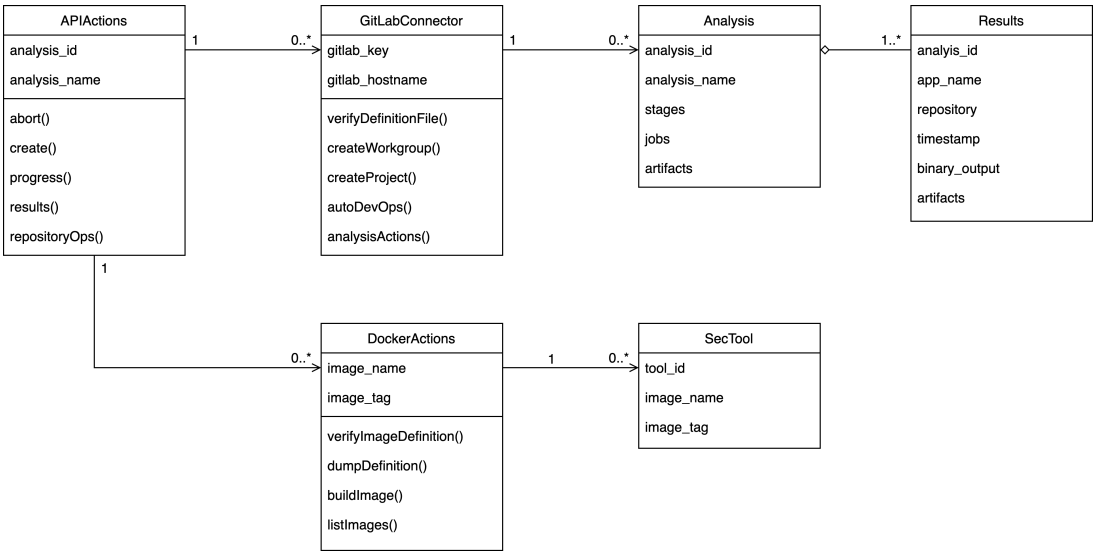


Figure 5.14: SecureApps@CI classes' diagram

The class `APIActions` takes care of the main operations requested through the SecureApps@CI API, specifically create or abort security analysis, check their progress or results and execute repository operations. The class `GitLabConnector` is encharged of GitLab CI API operations, like verify the definition form file, create a workgroup plus a project, and enabling the *Auto DevOps*. Next, we have a class called `Analysis` to hold security analysis. The `Analysis` has the following main attributes: id, name, stages, jobs, and respective artifacts of each security analysis pipeline. For each pipeline run cycle, another class called `Results` includes id, name, repository, timestamp, a binary output as simplified results, artifacts of the jobs, and a detailed report of the security tools that ran in that pipeline. There is an additional class called `DockerActions`, used to create or view details of a specific security tool from a list of existent Docker images with already built tools. A correspondent class is the security tool that has an id, name, tag of Docker images of tools.

All components from Figure 5.14 were encapsulated as a single Python Application delivered to customers. This Application is defined in a Docker image file called `Dockerfile`, available online at GitLab[40]. This `Dockerfile`[41] is a text document containing all commands a user needs to call to assemble an image in a CLI. The Docker image, based on a `python:3.7-alpine` image, has all OS packages, and all Python dependencies included. The `Dockerfile` is a multi-stage image. On the first part of the image, certain operations like installing packages in the alpine OS are done to run as `root`, but the remaining operations run as a custom user. The final image has a custom user with the number 9999 that runs the Python Application.

[40] https://gitlab.com/secureapps-ci/api_server/-/blob/master/Dockerfile
[41] https://docs.docker.com/engine/reference/builder/

Another important aspect is that the image runs safety[42], that examines the built dependencies for known security vulnerabilities using the open Python vulnerability database, called Safety DB. With this inspection, we assure that the Docker image will not have any known security vulnerability at the moment a SecureApps@CI instance is deployed.

On the application Docker image, we also have a health check inside the container of the API node to verify the status of the API service at a specific interval and timeout. The `HEALTHCHECK` statement tells Docker how to test a container to ensure that it is still operating as expected. Health tests can identify events such as web servers trapped in an endless loop and cannot accommodate new connections, even if the server process is still working.[43] The example in Code Snippet 5 shows health check inside a Docker file. In our case, we use `wget` instead of `curl`, because it is already built in on `Alpine Linux` Docker image, and it is used by its package manager `apk` despite the official Docker documentation showing an example with `curl` 6.

```
$ cat Dockerfile
HEALTHCHECK --interval=5s --timeout=3s CMD \
        wget -nv -t1 --spider https://localhost:5555/health || exit 1
```

Code Snippet 5: Dockerfile - health check with wget

```
$ cat Dockerfile
HEALTHCHECK --interval=5s --timeout=3s CMD \
        curl -f https://localhost:5555/health || exit 1
```

Code Snippet 6: Dockerfile - health check with curl

[42] https://github.com/pyupio/safety/
[43] https://docs.docker.com/engine/reference/builder/#healthcheck

CHAPTER 6

Evaluation and Results

In the current chapter we present the proof of concept scenarios, analyze if we fulfill the defined requirements, and summarize our solution's potentialities and results. The main goal of SecureApps@CI is ensuring that containerized applications have no dangerous components, providing a sanitizing process to ensure that these applications will run free of known vulnerabilities at the moment of their deployment. We will perform an intrinsic analysis of the application and not extrinsic, being the application's execution environment excluded from the analysis. We did not test the integration with commercial or enterprise security tools related to compliance or improving existing security policies. Nevertheless, it should be possible to integrate them if they have CLI or API capabilities. The tests chosen for our PoC include relevant open-source tools related to secret detection, dependency scanning, static and dynamic analysis and container's image scanning.

This chapter has three main sections, the first presents the evaluation scenarios, which initially show how safety evaluation tools behave locally in an isolated environment. Then we show the same safety tool running within SecureApps@CI, the respective definition of the analysis, and the analysis results. In the following section, we list and discuss how we fulfill the requirements defined in the architecture. In the final section, we summarize the potentialities and results accomplished with our solution.

6.1 PoC evaluation

SecureApps@CI was conceived to integrate various types of security evaluation tools to detect known security problems within the applications before dropping and launching them into production. To avoid the same issue repeatedly, it is also possible to create white lists or ignore specific tests' phases if approved by the organizations' business or security policies. To prove the solution's effectiveness, we will show the results from security tools running locally, and afterward, the exact same security tools integrated into our solution, where they pop up the same results. To incorporate various tools inside a security analysis pipeline, we program

a series of tests inside the security analysis definition form used by SecureApps@CI. Moreover, there is the possibility of passing to a second security tool test if the other chosen tool passes their test.

To evaluate our PoC implementation, we established five different scenarios using four different security tools. These security tools have distinct purposes: detecting secrets in the source code, static analysis of the application code, dynamic analysis of a running application and external analysis of source code. This section will demonstrate the different scenarios where a security tool runs isolated, presents its results, or runs inside a security analysis pipeline, managed by SecureApps@CI returning the same result. To increase the readability of this chapter, all file samples and results of the execution of security tools are presented in Code Snippets in Appendix 7.1.

6.1.1 Secret detection scenario

In this scenario, the objective is to demonstrate our system's effectiveness in detecting secrets. We refer to secrets as hard-coded or embedded credentials such as passwords, authentication tokens, and private keys. Given a sample code of an application with embedded credentials and a secret detection tool, we expect to find credentials with the tool alone and exact the same credentials with that tool running inside the SecureApps@CI system.

We start the secret detection scenario PoC, by using a secret detection tool, first isolated and afterwards running as part of a security analysis pipeline. The secret detection tool chosen is `truffleHog`[1] by *@dxa4481* and the target application is a subset of the `repo-supervisor` repository by Auth0[2].

Secret detection tool - running isolated

First, we run the secret detection tool alone at localhost, and observe the output. The target application includes sub-directories containing files with sample hard-coded secrets inside as we can observe in the file `authentication.urls.json` (see Code Snippet 12). It has an Uniform Resource Locators (URLs) with authentication parameters including user and password for FTP, HTTP, and HTTPS schemas. After running `truffleHog` locally as shown in Code Snippet 13, this tool detects credentials inside the file pointing out a `Password in URL` warning. It also tells us other useful information like branch, commit message, commit hash, date, difference from previous commits, path of the target file and strings found in the contents.

Secret detection tool - within SecureApps@CI

Next, we defined a security analysis pipeline using `truffleHog` and ran it in our system to verify that we get the exact same results from the secret detection tool used. For that, we

[1] https://github.com/dxa4481/truffleHog
[2] https://auth0.com

defined a security analysis form in the Code Snippet 7, add it to the repository, commit and push the YAML Ain't Markup Language (YAML) definition file, and afterward, the security analysis pipeline will be triggered if *Auto-DevOps* is enabled.

```
variables:
  APP_NAME: "repo-supervisor"
  GROUP_NAME: "frontend-team"
  GIT_URL: "https://github.com/auth0/repo-supervisor"
  BRANCH_NAME: "master"
  GIT_DEPTH: '4'
  ENV: "dev"

stages:
  - secret_detection

secrets:
  stage: secret_detection
  variables:
    SECRETS_RESULTS: 'secrets.json'
  script:
    - trufflehog ${CI_PROJECT_DIR}/ \
                 --json \
                 --regex \
                 --cleanup \
                 --entropy=False | tee ${CI_PROJECT_DIR}/${SECRETS_RESULTS} | jq -C
  artifacts:
    paths: ["${SECRETS_RESULTS}"]
    when: always
  allow_failure: false
```

Code Snippet 7: Secret detection - security analysis pipeline form

If *Auto DevOps* is disabled, a call from an HTTP client is enough to trigger a new security analysis pipeline and obtain the results as illustrated by the example with `curl` in the Code Snippet 14. Its `POST`[3] request needs three HTTP headers: `Content-Type` (indicates the media type of the resource), `Accept` (informs the server about the types of data that he can send back) and `X-API-KEY` (a token required for authorization). Other option was to use the SecureApps@CI API within a web browser.

After requesting a security analysis with a security tool for secret detection, the GitLab web interface can show the pipeline stages, each stage's progress, and possible dependencies among them. It also shows the different jobs' status, details, and artifacts of security tools related to each job. Code Snippet 15 shows a general overview of a secret detection inside a security analysis pipeline. Here we can see that after having the `docker+machine` executor, preparing the environment, and getting the source code from the Git repository, the `step_script` starts. In this job, first the packages `jq` and `git` are installed, then `truffleHog` is also installed. Finally, `truffleHog` runs and finds exactly the same we saw before.

This way, we can conclude that our solution can successfully spawn the `truffleHog` analysis. Another important factor is that the pipeline fails with error (`ERROR: Job failed: exit code 1`) and the appropriate Boolean output, that is, one (1), since the secret detection

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

tool found a credential hard-coded.

6.1.2 Static analysis scenario

In this scenario, the objective is to prove our system's effectiveness in performing a static analysis of the application source code. Given a sample of a deliberately vulnerable application and a SAST tool, we expect to discover vulnerabilities with the tool alone and exact the same vulnerabilities with that same tool running inside the SecureApps@CI system.

We start the PoC of the static analysis scenario, by using a SAST tool, first isolated and afterwards running as part as the security analysis pipeline. The tool chosen was `Scan (skæn)`[4] by ShiftLeft[5]. and the target application is deliberately vulnerable `Django`[6] application called `django.nV` [7] provided by `nVisium`[8]. The target application has an outdated version of Django, as we can observe in the example file called `requirements.txt` (see Code Snippet 16).

SAST tool - running isolated

First we execute `Scan (skæn)` (see Code Snippet 17), which shows that it detects Django version 1.8.3, an outdated version with several well - known vulnerabilities, described in CVE entries. Among them, one has a critical severity and severity score of 9.8 (out of 10), namely, the `CVE-2019-19844`[9]. It also shows a security scan summary where we can observe other useful information, like Python dependency scan, Python source analyzer, and Python security analysis of the target application.

SAST tool - within SecureApps@CI

First, we defined a security analysis pipeline and ran it in our system to verify that we get the same results from the tool `Scan (skæn)`. For that, we established a security analysis form in `Security-Analysis.gitlab-ci.yml` (see Code Snippet 8), add it to the repository, commit and push the YAML definition, and later, the security analysis pipeline will be triggered if *Auto-DevOps* is enabled. If *Auto-DevOps* is disabled, a call from an HTTP client is sufficient to unleash a new security analysis pipeline and obtain the results.

Then, we do an API call to SecureApps@CI with HTTP client `curl` with the command shown in Code Snippet 18. This `POST` request needs three HTTP headers: `Content-Type` (indicates the media type of the resource), `Accept` (informs the server about the types of data that he can send back) and `X-API-KEY` (a token required for authorization). API keys[10] are sent by the clients of our solution as a request header to make API calls. Notice that the

[4] https://slscan.io
[5] https://www.shiftleft.io
[6] https://www.djangoproject.com
[7] https://github.com/nVisium/django.nV
[8] https://www.nvisium.com
[9] https://nvd.nist.gov/vuln/detail/CVE-2019-19844
[10] https://swagger.io/docs/specification/authentication/api-keys/

```yaml
variables:
  APP_NAME: "djangonv"
  GROUP_NAME: "frontend-team"
  GIT_URL: "https://github.com/nVisium/django.nV.git"
  BRANCH_NAME: "master"
  GIT_DEPTH: '1'
  ENV: "dev"

stages:
  - static_analysis

sast:
  stage: static_analysis
  image:
    name: shiftleft/sast-scan
  script:
    - scan --src ${CI_PROJECT_DIR} \
           --type depscan,python \
           --out_dir ${CI_PROJECT_DIR}/reports
  rules:
    - when: always
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - $CI_PROJECT_DIR/reports/
    when: always
  allow_failure: false
```

Code Snippet 8: SAST - security analysis pipeline form

`api_key` is read as an environment variable, to prevent someone with access to the terminal history acquires the key in plain text.

Following a security analysis request, including a SAST security tool, the GitLab web interface can show the pipeline stages and each stage's progress. It also shows the different job statuses, details, and artifacts of security tools related to each job.

Finally, we show a general overview of `Scan (skæn)` (see Code Snippet 19) running inside a security analysis pipeline and the detection of the same problems previously identified.

6.1.3 Dynamic analysis scenario

In this scenario, the objective is to demonstrate our system's effectiveness in performing a dynamic analysis of a running application. Given a sample of a deliberately vulnerable application and a DAST tool, we expect to discover vulnerabilities with the tool alone and exact the same vulnerabilities with that same tool running inside the SecureApps@CI system.

We start the PoC of the dynamic analysis scenario, by using a DAST tool, first isolated and afterward running as part as the security analysis pipeline. The tool chosen was `Zed Attack Proxy (ZAP)`[11] by OWASP[12] and the target application was the deliberately insecure

[11] https://www.zaproxy.org/
[12] https://owasp.org/

web application `webgoat`[13], maintained by OWASP.

DAST tool - running isolated

First, we create a Bash script called `zap-quick-scan.sh` (see Code Snippet 20) that will run the `ZAP` with the required parameters and options to initiate the scan without user interaction. The `zap-cli`[14] is CLI tool for controlling `ZAP`. Essentially, the script will run a series of scans of a target URL. To run without user interaction, we need to disable the ZAP api_key. We define the high filter option as the minimum alert level. The options applied to run the self-contained scan were: spider, ajax spider and recursive scan. The scan includes sqli, xss, xss_reflected and xss_persistent tests.

After creating the `zap-quick-scan.sh` script, we ran a testing environment with `ZAP` against `WebGoat` as described in the `Compose file`[15] called `docker-compose.yml` (see Code Snippet 21). We define the deployment with `Compose`[16], a tool for specifying and spinning various programs within containers using a YAML file to define the application's services. Then, with a single command `docker-compose up -d` we create and start all services as shown in Code Snippet 22).

DAST tool - within SecureApps@CI

First, we defined a security analysis pipeline and ran it in our system to verify that we get the exact same results from the tool `ZAP` running isolated. For that, we established a security analysis form in `Security-Analysis.gitlab-ci.yml` (see Code Snippet 9). Then we add the security analysis form to the repository, commit and push the YAML definition file, and afterward, the security analysis pipeline will be triggered if *Auto-DevOps* is enabled. If *Auto-DevOps* is disabled, an HTTP client call is sufficient to unlock a new security analysis pipeline and obtain the results. Notice that the service `docker:dind` is mandatory since we will run three containers within Docker and the GitLab runner that requests this job needs access to Docker daemon to launch them.

Next, we do an API call to SecureApps@CI with HTTP client `curl` with the command shown in Code Snippet 23. This `POST` request needs three HTTP headers: `Content-Type` (indicates the media type of the resource), `Accept` (informs the server about the types of data that he can send back) and `X-API-KEY` (a token required for authorization). API keys[17] are sent by the clients of our solution as a request header to make API calls. Notice that the `api_key` is read as an environment variable to prevent someone with access to the terminal history acquires the key in plain text.

Following a security analysis request, including a DAST security tool, the GitLab web interface can show the pipeline stages and each stage's progress. It also shows the different

[13]https://github.com/WebGoat/WebGoat
[14]https://github.com/Grunny/zap-cli
[15]https://docs.docker.com/compose/compose-file/
[16]https://docs.docker.com/compose/
[17]https://swagger.io/docs/specification/authentication/api-keys/

```
variables:
  APP_NAME: "zap-webgoat"
  GROUP_NAME: "samples"
  GIT_URL: "https://gitlab.com/secureapps-ci/samples/zap-webgoat"
  BRANCH_NAME: "master"
  GIT_DEPTH: '5'
  ENV: "dev"

image: tmaier/docker-compose:17.03
services:
  - docker:dind

stages:
  - test

test:
  stage: test
  variables:
    ENV: dev
    TEST_TARGET: "http://webgoat:8080/WebGoat"
    DAST_REPORT_FILE: "dast_report.txt"
  script:
    - echo $PWD
    - docker-compose up -d webgoat webwolf
    - docker-compose run -d -e ENV=dev \
                    -e REPORT_FILE="$DAST_REPORT_FILE" \
                    -e TARGET="$TEST_TARGET" zap2docker
    - docker logs zapwebgoat_zap2docker_run_1 -f
  rules:
    - when: always
  artifacts:
    paths:
      - $DAST_REPORT_FILE
    when: always
  allow_failure: false
```

Code Snippet 9: DAST - security analysis pipeline form

job status, details, and artifacts of security tools related to each job.

Finally, we show a general overview of ZAP (see Code Snippet 24) running inside a security analysis pipeline and the detection of the same problems previously identified.

6.1.4 External analysis scenario

In this scenario, the objective is to prove our system's effectiveness in performing an application's external analysis. Given a sample of a deliberately vulnerable application and a tool to request external SAST analysis, we expect to discover vulnerabilities requesting the external analysis isolated and exact the same vulnerabilities requesting that analysis inside the SecureApps@CI system.

The external analysis scenario uses an external tool to analyze code quality and security, first isolated and afterward running as part as the security analysis pipeline. The external analysis

tool chosen is `SonarCloud`[18] by SonarSource[19] and the target application is a deliberately insecure web application `HelloShiftLeft`[20] maintained by ShiftLeft[21].

External tool - running isolated

In this scenario we will request an external analysis tool in a local machine. First, we create a property file called `sonar-project.properties` 25 that will define project sources, programming language type, language version, and location of the binaries. In this concrete case, the project key is `secureapps-ci_hello-shiftleft`, the organization is `secureapps-ci`, the project name is `hello-shiftleft`, sonar sources are located at `src/main/java`, programming language is `java`, language version is `1.8` and binaries located at `target/classes`. With this file, the sonar scanner CLI is able to determine all properties of our project and proceed with a proper and adequate analysis. We also need to update our `pom.xml` file with some properties like `java.version`, `projectKey`, and `organization` 26. After running `sonar scanner`[22] as shown in this example 28, we can observe in `sonarcloud.io`[23] that the application has six security hotspots related with `Insecure configuration`. The remediation suggestion is to make sure that the debug feature is deactivated before delivering the code in production.

External tool - within SecureApps@CI

First, we will define a security analysis pipeline and run it in security solution to prove that it does not change the output results of the external analysis tool `SonarCloud`. For that, we establish a security analysis form called `Security-Analysis.gitlab-ci.yml` 10, add it to the repository, commit and push the YAML definition, and later, the security analysis pipeline will be triggered if *Auto-DevOps* is enabled. If *Auto-DevOps* is disabled, a call from an HTTP client is sufficient to unlock a new security analysis pipeline and obtain the results.

After having the form defined, we do an API call to SecureApps@CI with HTTP client `curl` with the command shown in Code Snippet 29. This `POST` request needs three HTTP headers: `Content-Type` (indicates the media type of the resource), `Accept` (informs the server about the types of data that he can send back) and `X-API-KEY` (a token required for authorization). API keys[24] are sent by the clients of our solution as a request header to make API calls. Notice that the `api_key` is read as an environment variable to prevent someone with access to the terminal history acquires the key in plain text.

Following a security analysis request, including external analysis tool `SonarCloud`, the GitLab web interface can show the pipeline stages and each stage's progress. It also shows the different job status, details, and artifacts of security tools related to each job.

[18] https://sonarcloud.io
[19] https://www.sonarsource.com
[20] https://github.com/ShiftLeftSecurity/HelloShiftLeft
[21] https://www.shiftleft.io
[22] https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/
[23] https://sonarcloud.io/project/security_hotspots?id=secureapps-ci_hello-shiftleft
[24] https://swagger.io/docs/specification/authentication/api-keys/

```yaml
variables:
  APP_NAME: "HelloShiftLeft"
  GROUP_NAME: "samples"
  GIT_URL: "https://github.com/ShiftLeftSecurity/HelloShiftLeft.git"
  BRANCH_NAME: "master"
  GIT_DEPTH: '2'
  ENV: "qa"

stages:
  - build
  - external_analysis

build-app:
  stage: build
  image:
    name: maven:3-jdk-11
    entrypoint: [""]
  script:
    - mvn clean package
  artifacts:
    paths:
      - target/

sonarcloud-check:
  stage: external_analysis
  needs: ["build-app", "secrets"]
  image:
    name: sonarsource/sonar-scanner-cli:latest
    entrypoint: [""]
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
    GIT_DEPTH: "0"
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
    - sonar-scanner
  only:
    - merge_requests
    - master
    - develop
  dependencies:
    - build-app
```

Code Snippet 10: External analysis - security analysis pipeline form

Finally, we show a general overview of `SonarCloud` (see Code Snippets 30, 31) running inside a security analysis pipeline and the detection of the same problems previously identified.

6.1.5 Integration analysis scenario

In this scenario, the objective is to demonstrate our system's flexibility and adaptability by integrating various security assessment tools given an analysis definition form. We will select a set of analysis tools (secret detection, SAST and external analysis), run them in dedicated CI/CD pipeline, and, finally, collect their results. The secret detection tool is `truffleHog`, the static analysis tool is `Scan (skæn)` and external analysis is provided by `SonarCloud`. Given a sample of a deliberately vulnerable application and various security assessment tools, we expect to see the flow and sequence of analysis tools in the SecureApps@CI system precisely as defined in the analysis definition form.

We start by making an API call to SecureApps@CI API within an HTTP client called `curl` in the command line as shown below 32. This `POST` request just needs three HTTP headers, namely, `Content-Type`, `Accept`, and `X-API-KEY`. The `Content-Type` indicates the media type of the resource, `Accept` informs the server about the types of data that he can send back, and `X-API-KEY` is a token required for authorization.

We can define an orchestration flow, integrate different security tools, pass dependencies between jobs of different stages, and ensure that external analysis only happens if there is no private or sensitive information inside the source code. The GitLab web interface can show the pipeline stages, each stage's progress, and possible dependencies between them. It also shows the different job statuses, details, and artifacts of security tools related to each job.

We established a security analysis form called `Security-Analysis.gitlab-ci.yml` 11 and added it to the repository. This form has secret detection, SAST and external analysis inside. The Pipeline has four stages `build`, `secret_detection`, `static_analysis`, and `external_analysis`. The `static_analysis` and is allowed to fail since we have a second SAST done by `external_analysis`. The `external_analysis` depends on the `secret_detection job` called `secrets` as we can observe by the key and value `needs: ["secrets"]`. This dependency aims to ensure that we only execute an external analysis if we do not find any secrets or sensitive information inside the source code.

This application used is the same as in the external analysis scenario. Its called `HelloShiftLeft` and provided by ShiftLeftSecurity[25]. On top of the demo application, we added the files needed for running the security analysis pipeline and external analysis by `SonarCloud`. The code of this application is available here[26]. The pipelines can be observed at *gitlab.com*[27]. The external analysis of the application can be viewed at *sonarcloud.io*[28].

In the security analysis pipeline definition form 11 we orchestrate three different security tools in an automated way by using SecureApps@CI. This analysis is similar to the previous external analysis scenario 6.1.4, since it has the same first and last stages `build` and `external_analysis`. It differs in the middle stages `secret_detection` and `static_analysis`. At these stages, for secret detection, we use `truffleHog` and for static

[25]https://www.shiftleft.io/
[26]https://gitlab.com/secureapps-ci/samples/hello-shiftleft
[27]https://gitlab.com/secureapps-ci/samples/hello-shiftleft/-/pipelines
[28]https://sonarcloud.io/dashboard?id=secureapps-ci_hello-shiftleft

application security testing `Scan (skæn)`.

Finally, the security analysis pipeline will be triggered if *Auto-DevOps* is enabled. If *Auto-DevOps* is disabled, a call from an HTTP client as shown here 32, is sufficient to unlock a new analysis and acquire the results. Observing the security analysis pipeline results of these security tools is possible with GitLab web interface 6.1.

We can see that it has four stages, build, secret detection, static, and external analysis. All stages passed successfully and have a green checkmark except the static analysis that has an exclamation mark with brown color. The static analysis stage is allowed to fail, so the pipeline continues with the external analysis. Static analysis fails due to dependency issues with the package `com.fasterxml.jackson.core:jackson-databind`, entity leaks in the file `AccountController.java:[lines 21-71]`,[29] and the use of a non-recommended cryptographic hash function called API MD5 (MDX) in file `CustomerController.java:[lines 73-388]`[30]. Furthermore, since the secret detection did not find any secrets or private information, the stage passes successfully, and the external analysis executes. If the secret detection job fails, GitLab CI skips the external analysis job.

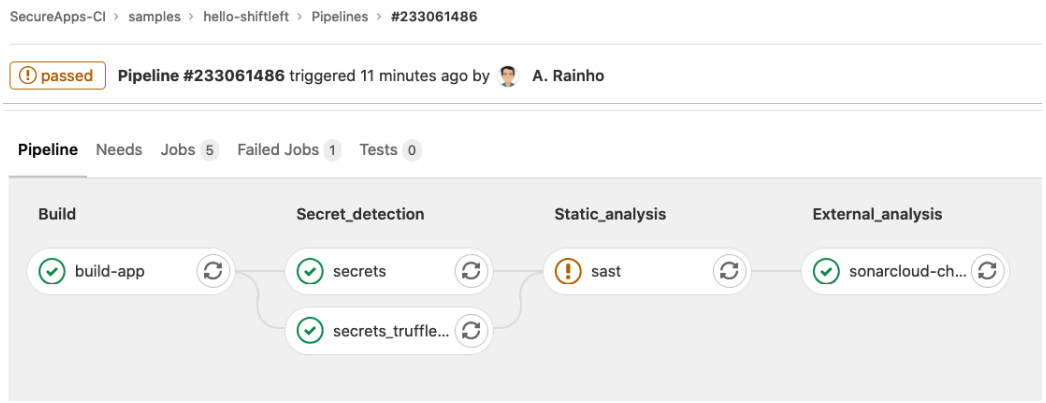


Figure 6.1: PoC - all security tools integration scenario

[29] `https://gitlab.com/secureapps-ci/samples/hello-shiftleft/-/blob/master/src/main/java/io/shiftleft/controller/AccountController.java`

[30] `https://gitlab.com/secureapps-ci/samples/hello-shiftleft/-/blob/master/src/main/java/io/shiftleft/controller/CustomerController.java`

```
variables:
  APP_NAME: "HelloShiftLeft"
  GROUP_NAME: "samples"
  GIT_URL: "https://github.com/ShiftLeftSecurity/HelloShiftLeft.git"
  BRANCH_NAME: "master"
  GIT_DEPTH: "0"
  ENV: "dev"

stages:
  - build
  - secret_detection
  - static_analysis
  - external_analysis

build-app:
  stage: build
  image:
    name: maven:3-jdk-11
  script:
    - mvn clean package
  artifacts:
    paths:
      - target/

secrets:
  stage: secret_detection
  image:
    name: python:3-alpine
  script:
    - apk add jq git
    - python -m pip install --upgrade truffleHog
    - trufflehog --json --regex --cleanup file:///${CI_PROJECT_DIR}/ \
                 --entropy=False | tee ${CI_PROJECT_DIR}/${SECRETS_RESULTS} | jq -C
  allow_failure: false

sast:
  stage: static_analysis
  image:
    name: shiftleft/sast-scan
  script:
    - scan --src ${CI_PROJECT_DIR} \
           --type depscan,python  --out_dir ${CI_PROJECT_DIR}/reports
  allow_failure: true

sonarcloud-check:
  stage: external_analysis
  needs: ["build-app", "secrets"]
  image:
    name: sonarsource/sonar-scanner-cli:latest
  script:
    - sonar-scanner
  dependencies:
    - build-app
```

Code Snippet 11: Diversity of tools - security analysis pipeline definition form

6.2 Achieved requirements

The main goal of our system was to facilitate the detection of vulnerabilities in containerized CI/CD applications prior to their deployment. In an SDLC process, applications developed and disseminated from a CI/CD usually do not have quality processes focusing the detection of elements that may cause potential security threats. Our SecureApps@CI system was meant to be a functional solution with flexible and integration capabilities, capable of running various security tools arbitrarily to achieve this goal. We start by evaluating the requirements proposed and how they were achieved. The requirements were analysis definition, components, deployment, architecture, availability, results and report, web interface, version control system, and client requirements.

Analysis definition

For achieving the analysis definition requirement, a security analysis pipeline is declared in a form describing the desired security analysis and sent over to our system as a file. The only file format allowed is YAML format, readable by both computers and humans. As we have shown in the different scenarios of PoC evaluation section, we may use and adapt the analysis definition form to run various security tools arbitrarily. In this form, we must declare the application to be analyzed and correspondent details: the branch, environment, source code repository and team. It is possible to add more parameters to determine how, when, and where each tool will run, as well as special conditions such as outbuildings or white lists.

Components

To reach the components requirement, we implemented our system with open-source software. We used Python[31] as programming language, Docker[32] for deployment, and GitLab community edition[33] for CI/CD and security analysis pipelines. For testing and implementating PoC scenarios, all security tools included in security analysis pipelines were constructed using open-source container images from Docker Hub[34]. Docker uses an open-source tool, called `BuildKit`[35], which takes instructions from a Dockerfile and constructs a Docker image. The security tools used were `truffleHog`, `scan` and `ZAP`. The API contract was developed and defined with Swagger, a tooling ecosystem for developing APIs with OAS[36]).

[31] https://python.org/
[32] https://www.docker.com/
[33] https://about.gitlab.com/install/ce-or-ee
[34] https://hub.docker.com
[35] https://github.com/moby/buildkit
[36] https://www.openapis.org/faq

Deployment

To meet the deployment requirement, our system uses a container engine for launching active components, namely, Docker engine. The construction of active components employs minimal container images, including just applications and dependencies. The usage of Dockerfile[37] allows the declaration of the build instructions to produce the final image. A Dockerfile is a text file that describes how to assemble a private filesystem for a container, and contains metadata describing how to run a container based on this image.

We went even further by using the Multistage[38] build functionality, which allowed us to create images of smaller containers with improved caching and a reduced security footprint. As shown in Code Snippet 33, Multistage allows building and installing all dependencies in the first stage, and in the second stage uses only the needed binaries and libraries already produced in the first stage. The final version of the `Dockerfile` can be observed at `gitlab.com/secureapps-ci`[39].

We used the `pip` package manager to install and manage additional libraries or dependencies not distributed in the standard Python library. All libraries and dependencies are listed in a `requirements.txt` file that can be observed in `gitlab.com/secureapps-ci`[40]. The pip packages are pinned to ensure that our Python application and correspondent dependencies for production are built in a predictable and deterministic fashion. Before each application runs inside a container, a static analysis is performed to ensure we do not run any vulnerable dependencies. After a successful build of the components, a container engine, in our case Docker engine, executes their deployment.

Architecture

The architecture is modular and extensible. Each block has focused functionality, consistent interfaces, and contains one or more components that fulfill this requirement. The blocks that have boundaries with others provide a service exposed through an API. We also assure that blocks interacting and communicating with others can do it consistently. The exceptions are the block that shows the results (Visualizer) and the agent nodes that run the security analysis pieces. Our system is interoperable and flexible to attach or detach new components inside each block, if needed. This flexibility is possible since we follow the principles of a microservices architecture, by designing each system component as software application suites of independently deployable, loosely coupled, collaborating services.

Our system has four major blocks, as referred in Figure 4.4, they are Receiver, Nucleus, Orchestrator, Visualizer. These blocks became API server, backends, GitLab CI, runners, and Docker executor in the implementation phase as shown in Figure 5.1. In our context, GitLab-CI plays the roles of orchestrator and visualizer. Both API server and backends

[37] https://docs.docker.com/engine/reference/builder
[38] https://docs.docker.com/develop/develop-images/multistage-build
[39] https://gitlab.com/secureapps-ci/api_server/-/blob/master/Dockerfile
[40] https://gitlab.com/secureapps-ci/api_server/-/blob/master/requirements.txt

were developed in Python and are deployed on the same container. The various components need to be connected and communicate for the system to work, and they all run inside containers. Our system uses three distinct APIs, specifically the SecureApps@CI API, GitLab CI and Docker engine API. The SecureApps@CI API was implemented according to OAS.

Availability

SecureApps@CI is available for CI/CD pipelines or developers without plugins or additional packages. It has an API that works both for developers or CI/CD pipelines. Making requests with one HTTP client from a source code repository can create a new security analysis on a target application. We developed the API using Swagger tools to help generate and maintain the API documentation, ensuring that up-to-date as the API contract evolves. The API definition is in YAML and has all schemas, endpoints, requests, responses, and correspondent variables.

Results and report

The analysis results need to include a Boolean output and a detailed report. To accomplish this requirement, the analysis results provided by an analysis pipeline have a `pass` or `fail` generic outcome and a detailed report for each security tool used. The report format depends on the security tools' output. We give preference to formats simple to read by humans and easy to analyze by machines such as JSON or, when it is not possible, tables or text formats.

Web interface

The provided analysis results - the Boolean output and a report - are visible through a web interface. Our CI/CD system (GitLab) includes a web interface that makes it possible to view the binary output and a detailed security analysis report to meet this requirement. On pipelines with stages that depend on other stages (thus implementing a series analysis), GitLab generates a dependency graph visible in the User Interface (UI). It is possible to view if an analysis pipeline passed or failed, and the detailed report output of each security tool has Figure 6.2 highlights.

The running pipelines have a blue color and the ones that passed with errors are brown. The gray pipelines are the ones canceled and the green pipelines are the ones that passed successfully. Finally, in red, we have the pipelines that failed. We have a pizza circle with the progress for the pipelines that passed. When the pipelines pass with errors, the symbol is an exclamation mark and a backslash symbol for the canceled pipelines. If the pipelines fail, there will be a cross symbol. It is also possible to view the list of jobs plus their execution details, and the list of failed jobs plus the correspondent details. It is also possible to view the list of jobs plus their execution details and the list of failed jobs plus the correspondent

details.

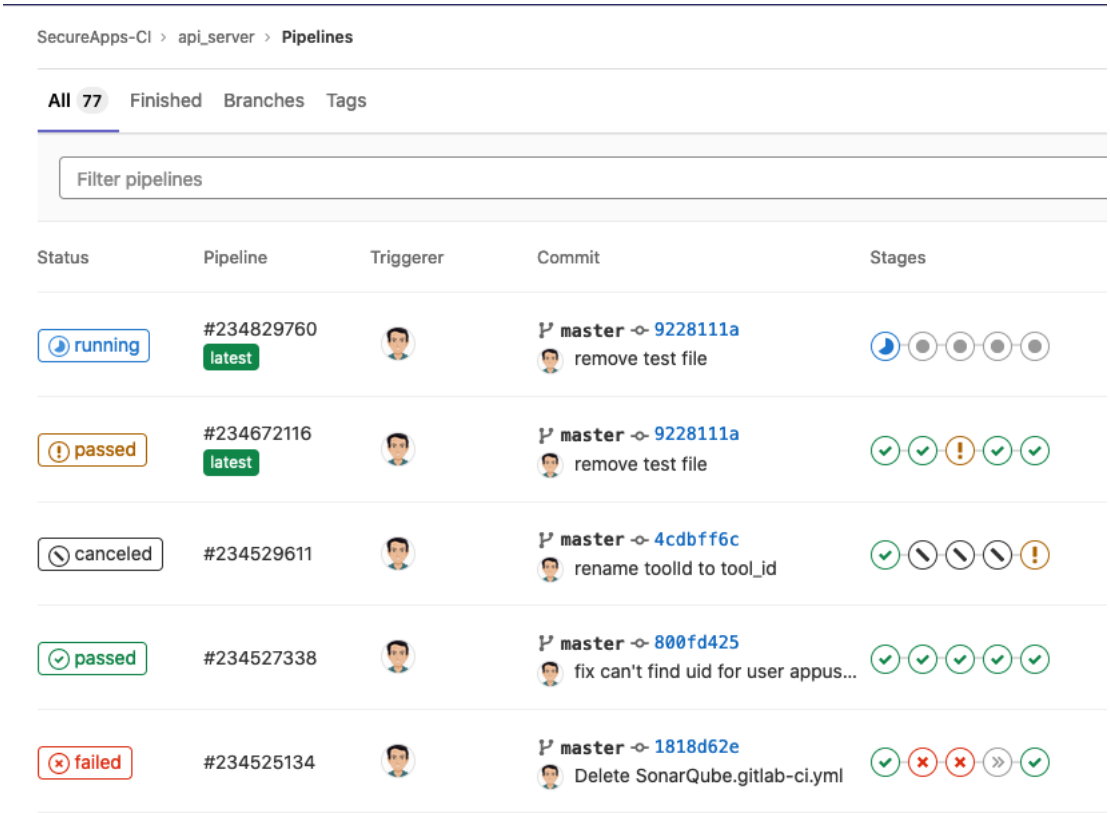


Figure 6.2: Achieved requirement - web interface

Version control system

The Distributed Version Control System used was Git. Its usage is a strict requirement, given the context of our system and the organizations' environment in our analysis context. Consequently, CI/CD Pipelines employ Git as the only version control system. This requirement was attained by having the source code of applications subject to security analysis on Git repository. Furthermore, GitLab CI has a native support of Git, and the security analysis pipelines depend on it.

Client requirements

Clients of the SecureApps@CI must have an HTTP client. Developers' machines or autonomous services need an HTTP client installed to manage security analysis pipelines. Clients also need Git (that derives from the previous requirement). We improved the compatibility and consequently fostered the adoption of our system by reducing the client requirements to a bare minimum. The potential clients that we have tested were CLI clients curl[41], wget[42],

[41] https://curl.se/
[42] https://www.gnu.org/software/wget/

httpie[43], Python client SDK[44], and Postman App[45]. In all these clients, it is possible to call the API endpoints and receive the correspondent responses. The API also provides a UI called Swagger UI[46] to visualize and interact with the API resources from a Web Browser. This UI presents visual documentation for client-side consumption and the ability to make requests and receive responses from the API.

6.3 Final thoughts

We can state that our SecureApps@CI proposal can easily integrate several security analysis tools. Security tools do not lose effectiveness inside our system, so if a tool works in isolation, the same tool will work in our environment. Security tools can be open-source or commercial, although we just ran tests with open-source tools. All open-source tools used had a CLI or anAPI. The code of these tools was not modified, so we could probably integrate commercial tools if they also had a CLI or an API. If it were a commercial tool with only a web interface, it would be more complex and time-consuming. However, it would be possible with Selenium[47], that enables and supports web browser automation.

We tested several validations flows in the various scenarios presented. The more complex flows had dependencies between the various stages of the analysis pipeline. The external analysis scenario 6.1.4 and integration analysis scenario 6.1.5 demonstrated the system ability to perform tasks or not according to the previous stages' outcome. We could make an orchestration that does not pass for the next test if any previous test had failed. The orchestration process is valuable, since it allows a controlled execution according to several dependencies. It also showed us that it is possible to use artifacts from a previous phase at a later stage, useful if we need to build an application before running a specific analysis, like the case of our external analysis scenario 6.1.4. These interactions and inter-dependencies are crucial, since an organization usually intends to use an external analysis only if the secret detection tests performed in a preliminary phase passes; otherwise, it skips that external analysis.

[43] https://httpie.io/
[44] https://swagger.io/tools/swagger-codegen/
[45] https://www.postman.com/
[46] https://swagger.io/tools/swagger-ui/
[47] https://www.selenium.dev

CHAPTER 7

Conclusion

The rise of the DevOps movement is associated with major changes in software development. DevOps created or drove the use of new technologies such as cloud, containers, serverless, and open-source software in general. DevOps' agility and speed bring new challenges, and securing the DevOps process is vital. The innovative movement DevSecOps answers these needs by including application security principles in a typical DevOps cycle, delivering security best practices earlier in SDLC. Organizations need to secure DevOps' workflows by enabling vulnerability scanning early in CI/CD pipelines and considering application security from the start.

The purpose of this work was to reduce the impact of known security vulnerabilities in microservices by examining the images and containers associated, in order to provide a safe collection of microservices deployed by CI/CD pipelines. This way, the system we proposed, SecureApps@CI, can integrate various security tools to ensure that applications are clean and sanitized before releasing them in production environments, where they are available worldwide to the organization's customers.

We demonstrate that analysis tools' execution results do not change when used in our system. We validate that our system allows the definition and orchestration of several interlinked analysis flows. SecureApps@CI is a flexible and adaptable system that selects a set of analysis tools, runs them in dedicated CI/CD pipelines, and finally collects their results. This system allows programming and orchestrating a series of security tests through a definition form to select several security analysis tools and according to the defined dependencies.

The creation of the system SecureApps@CI gives the organizations a tool for performing arbitrary security analysis of applications at the request of developers or CI/CD pipelines. The creation of a REST API provides this capability to call SecureApps@CI as quickly as possible, with a minimum of dependencies and reducing friction in SDLC by software developers or CI/CD pipelines. Other advantages for organizations include open-source software; version control hosted on-premises or in the cloud and adaptability to different CI/CD systems. Our system also runs the various security assessment tools in isolated, encapsulated, or containerized environments.

7.1 Future work

In terms of possibilities for future work, we have identified four topics: the introduction of OAuth[1], implementation of a health status check for the API node, the addition of tags for applications' repositories, and whitelist annotations.

Secure security definition

The API accepts credentials (API keys) transported over the network. These credentials are sent over the network on each API call repeatedly and are exposed to attack attempts to retrieve the keys. Changing to a more secure security definition like OAuth 2.0[2] is a possible remediation. OAuth is an open protocol to allow secure authorization. It uses access tokens with limited lifetime and authorizations (scope) granted by the Resource Owner from an authorization server. For determining security issues, online tools like API contract security audit[3] by 42Crunch[4] can be used to obtain a detailed report of potential vulnerabilities and other issues on the API contract.

Health status of API node

The API Node Dockerfile definition should have reliable and precise health checks [5]. Usual health checks are available at endpoints like `/api/v1/ping`, `/api/v2/status`, `/api/v3/health` give developers and Services the answer if APIs are available and healthy. Several steps need to be implemented to have consistent health checks. At this moment, we have a simple health check inside the container of the API node that verifies if a `/health` endpoint is alive. Besides this, other checks are required to ensure that the backend processes requests properly, the orchestrator can communicate with runners and the container engine can launch security analysis tools. With all this, we actually verify that a Security Analysis can actually run.

The number of replicas of the API Node should also be defined in a Docker Swarm service[6] or in case of Kubernetes Deployment by defining ReplicaSets[7] to ensure at least one replica of an API Node is running. Additionally, a tiny pipeline template should be triggered within a short time interval and the results collected. The tiny pipeline includes sample tools (in serial and parallel tests). If the result status is `"complete"` the health status response should be `"up"` otherwise the response is `"down"`.

[1] https://oauth.net/2/
[2] https://oauth.net/articles/authentication
[3] https://apisecurity.io/tools/audit
[4] https://www.42crunch.com
[5] https://mesosphere.github.io/marathon/docs/health-checks.html
[6] https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/
[7] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

Tags for application repositories

Like most VCSs, Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0, and so on).[8] Tags for application repositories are crucial, since developers and organizations rely on this feature to trigger pipelines, deployment on specific environments, quality assurance for certain versions, and to control the release of applications for clients. Here, the purpose could be a concrete tag that would trigger the security analysis pipeline, let us say, a tag `vs-final` activates a security analysis pipeline. A possible scenario could be to create this tag in a staging environment before the application is launched into production or create a tag `weekend-dast` in a repository where a security analysis pipeline will launch security tools in the weekend. The idea will be to create a specific tag to schedule tasks to run in the background at regular time intervals. Available options can be hourly, daily, weekly, monthly, and other custom times, such as weekdays (excluding Saturday and Sunday) or weekends.

Whitelist annotations

When using security analysis' tools, there are situations where there is a need to consider existing vulnerabilities as acceptable, in case the business takes the risk of releasing a version of a software product on the market, even knowing that it includes a security vulnerability. True positives are an example of this case, where they are whitelist when accepted and validated according to the organization policies for the true positives in question. Another possibility is false positives, often the case with static analysis tools. For ignoring true or false positives properly, it is essential to find appropriate ways to manage them and have the least possible impact on software development lines. For this to happen, our system needs to avoid ignoring a CI/CD pipeline job within a stage and write annotations in the security analysis form's definition (a YAML file). The pipeline job describes the security analysis tools chosen, error labels, and a name to ignore when finding a specific error. Some security analysis tools can ignore errors by adding comments before the code lines that originated the error in question. In our case, the annotations would be described and defined in the security analysis form itself.

[8] https://git-scm.com/book/en/v2/Git-Basics-Tagging

Bibliographic References

[1] S. Armstrong, "DevOps for Networking," in *DevOps for Networking*, 2016, ISBN: 9781786460561. arXiv: arXiv:1011.1669v3.

[2] M. Arntz, *Iterative Development vs Agile Development | ASPE*, 2019. [Online]. Available: https://aspetraining.com/resources/blog/iterative-development-vs-agile (visited on 12/20/2019).

[3] Atlassian, *What is DevOps? | Atlassian*, 2019. [Online]. Available: https://www.atlassian.com/devops (visited on 12/20/2019).

[4] AWS, *What is Cloud Computing*, 2019. [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/ (visited on 12/20/2019).

[5] A. Bettini, *Secure Container Development Pipelines with Jenkins*, 2017. [Online]. Available: https://www.cloudbees.com/sites/default/files/2016-jenkins-world-secure_container_development_pipelines_with_jenkins_1.pdf.

[6] EUR-Lex, *The general data protection regulation applies in all Member States from 25 May 2018*, 2018. [Online]. Available: https://eur-lex.europa.eu/content/news/general-data-protection-regulation-GDPR-applies-from-25-May-2018.html (visited on 01/21/2021).

[7] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," *Proceedings - 2nd International Conference on Multimedia Information Processing and Retrieval, MIPR 2019*, pp. 467–470, 2019. DOI: 10.1109/MIPR.2019.00094.

[8] Gartner, *Integrating Security Into the DevSecOps Toolchain*, 2019. [Online]. Available: https://www.techwire.net/sponsored/integrating-security-into-the-devsecops-toolchain.html (visited on 01/30/2021).

[9] A. Khan, "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, Sep. 2017, ISSN: 23256095. DOI: 10.1109/MCC.2017.4250933.

[10] Kharnagy, *File:Devops-toolchain.svg - Wikimedia Commons*, 2019. [Online]. Available: https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg (visited on 01/30/2021).

[11] N. MacDonald and I. Head, "DevSecOps: How to Seamlessly Integrate Security Into DevOps," *Gartner Research*, no. September, p. 15, 2016. [Online]. Available: https://www.gartner.com/doc/3463417/devsecops-seamlessly-integrate-security-devops.

[12] M. Meli, M. R. McNiece, and B. Reaves, "How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories," in *Proceedings 2019 Network and Distributed System Security Symposium*, Reston, VA: Internet Society, 2019, ISBN: 1-891562-55-X. DOI: 10.14722/ndss.2019.23418.

[13] P. Mell and T. Grance, "The NIST definition of cloud computing: Recommendations of the National Institute of Standards and Technology," in *Public Cloud Computing: Security and Privacy Guidelines*, 2012, pp. 97–101, ISBN: 9781620819821.

[14] D. Murugaiyan, "International Journal of Information Technology and Business Management WATEERFALL Vs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC," *International Journal of*

Information Technology and Business Management, vol. 2, no. 1, pp. 26–30, 2012, ISSN: 2304-0777. [Online]. Available: www.jitbm.com.

[15] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A Multivocal Literature Review," in *Communications in Computer and Information Science*, vol. 770, 2017, pp. 17–29, ISBN: 9783319673820. DOI: 10.1007/978-3-319-67383-7_2.

[16] S. Newman, *Building Microservices*. 2015, p. 280, ISBN: 978-1-491-95035-7. [Online]. Available: https://www.oreilly.com/library/view/building-microservices/9781491950340.

[17] O'Reilly, *The State of the Internet Operating System - O'Reilly Radar*, 2019. [Online]. Available: http://radar.oreilly.com/2010/03/state-of-internet-operating-system.html (visited on 12/20/2019).

[18] C. Paule, "Securing DevOps: detection of vulnerabilities in CD pipelines," Ph.D. dissertation, University of Stuttgart Universitätsstraße, 2018. [Online]. Available: http://dx.doi.org/10.18419/opus-10029.

[19] PCISSC, *Official PCI Security Standards Council Site - Verify PCI Compliance, Download Data Security and Credit Card Security Standards*, 2018. [Online]. Available: https://www.pcisecuritystandards.org/pci%7B%5C_%7Dsecurity/ (visited on 11/27/2020).

[20] B. Piper and D. Clinton, "Introduction to Cloud Computing and AWS," in *AWS Certified Solutions Architect Study Guide*. Indianapolis, Indiana: Wiley, Mar. 2019, ch. 1, pp. 1–19, ISBN: 9781119560395. DOI: 10.1002/9781119560395.ch1.

[21] S. Ragan, "Code Spaces forced to close its doors after security incident," *CSO, June*, vol. 18, 2014. [Online]. Available: https://www.csoonline.com/article/2365062/code-spaces-forced-to-close-its-doors-after-security-incident.html.

[22] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security Smells in Ansible and Chef Scripts: A Replication Study," Jul. 2019. arXiv: 1907.07159. [Online]. Available: http://arxiv.org/abs/1907.07159.

[23] T. Rangnau, R. V. Buijtenen, F. Fransen, and F. Turkmen, "Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines," in *Proceedings - 2020 IEEE 24th International Enterprise Distributed Object Computing Conference, EDOC 2020*, IEEE, Oct. 2020, pp. 145–154, ISBN: 9781728164731. DOI: 10.1109/EDOC49727.2020.00026.

[24] N. Relic, *SecDevOps: Injecting Security Into DevOps Processes*, 2019. [Online]. Available: https://blog.newrelic.com/technology/what-is-secdevops/ (visited on 12/20/2019).

[25] B. Sheairs, *[eBook] DevSecOps Lifecycle Adding Security to Agile | Synopsys*, 2020. [Online]. Available: https://www.synopsys.com/software-integrity/resources/ebooks/devsecops-lifecycle-champions.html (visited on 11/27/2020).

[26] Simpson and G. Bernard, *CI/CD Software Security Automation*. Oct. 2018. [Online]. Available: https://www.osti.gov/biblio/1594298.

[27] S. Singh and N. Singh, "Containers & Docker: Emerging roles & future of Cloud technology," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, IEEE, Apr. 2016, pp. 804–807, ISBN: 978-1-5090-2399-8. DOI: 10.1109/ICATCCT.2016.7912109.

[28] The Kubernetes Authors, *Kubernetes Components | Kubernetes*, 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

Appendices

API documentation

In this document, the SecureApps@CI solution REST API is described in detail.

Usage

get **/health**

Returns the API health status

Parameter

No parameters

Response application/json

200 ok

```
{
    "api_status": "up/down"
}
```

get **/version**

Returns the API version

Parameter

No parameters

Response application/json

200 ok

```
{
    "api_version": "v1"
}
```

post **/analysis/create**

Create a new Analysis according to the uploaded definition

Parameter

No parameters

Body application/octet-stream

```
Example values are not available for application/octet-stream
    media types.
```

Response application/json

400 The specified YAML definition is invalid (e.g. not a valid YAML)

401 API key is missing or invalid

404 The YAML definition for a new analysis was not found

200 ok

```
{
    "analysis_id": "string",
    "analysis_name": "string",
    "accepted": true
}
```

100 The Analysis request has been received and is processing, but no response is available

102 The Analysis is OK so far, client should continue the request or ignore the response if the request already finished

delete **/analysis/{analysisId}/abort**

remove a storage with id

Parameter

analysisId	The ID of the Analysis to abort

Response application/json

200 ok


```
{
    "analysis_id": "string",
    "abort_status": "string"
}
```

get **/analysis/{analysisId}/progress**

The Progress of Analysis Stages 1,2,3 and tool jobs

Parameter

analysisId	The ID of the Analysis to retrieve progress

Response application/json

200 ok


```
{
    "analysis_id": "string",
    "analysis_name": "string",
    "progress": "accepted",
    "stage1_id": 0,
    "stage1_status": "string",
    "stage2_id": 0,
    "stage2_status": "string",
    "stage3_id": 0,
    "stage3_status": "string",
    "laststage_id": 0,
    "laststage_status": "string"
}
```


400 The specified analysisId is invalid (e.g. not a number)

401 API key is missing or invalid

404 The analysis with the specified ID was not found

get **/analysis/{analysisId}/stage/{stageId}/jobs**

Returns Jobs for a stageId

Parameter

analysisId	The ID of the Analysis to retrieve Stages
stageId	The ID of the Stage to return Jobs

Response application/json

200 ok


```
{
    "stage_id": 0,
    "stage_name": "string",
    "job_list": [
        {
          "job_id": 0,
          "job_name": "string",
          "job_type": "string",
          "job_definition": "string"
        }
    ]
}
```


400 The specified analysisId or stageId is invalid (e.g. not a number)

401 API key is missing or invalid

404 The Analysis or Stage with the specified IDs was not found

get **/analysis/{analysisId}/results**

Returns results status of Analysis (can include report if complete)

Parameter

analysisId	The ID of the Analysis to retrieve results

Response application/json

200 ok


```
{
    "analysis_id": "string",
    "analysis_status": "string",
    "report_id": 0,
    "report_results": {}
}
```


400 The specified analysisId is invalid (e.g. not a number)

401 API key is missing or invalid

404 The analysis with the specified ID was not found

post **/tool/create**

Create a new Tool according to the uploaded definition

Parameter

No parameters

Body application/octet-stream

```
Example values are not available for application/octet-stream
    media types.
```

Response application/json

200 ok

```
{
    "tool_id": "string",
    "tool_name": "string",
    "accepted": true
}
```

400 The specified Dockerfile definition is invalid (e.g. not a valid Dockerfile)

401 API key is missing or invalid

404 The Dockerfile definition for a new Tool was not found

delete **/tool/{toolId}/remove**

Remove the specified Security Tool

Parameter

toolId The ID of the Tool

Response application/json

200 ok


```
{
    "tool_id": "string",
    "tool_name": "string",
    "remove_status": "string"
}
```


400 The specified toolId is invalid (e.g. not a number)

401 API key is missing or invalid

404 The Tool with the specified ID was not found

get **/tool/{toolId}/details**

Returns details of specific Security Tool

Parameter

toolId The ID of the Tool

Response application/json

200 ok


```
{
    "tool_id": "string",
    "tool_name": "string",
    "tool_status": "string",
    "dockerfile_id": "string",
    "dockerfile_dump": {}
}
```


400 The specified toolId is invalid (e.g. not a number)

401 API key is missing or invalid

404 The Tool with the specified ID was not found

get **/tools/list**

Collect available Security Tools

Parameter

toolId The ID of the Tool

Response application/json

200 ok


```
[
    {
    "tool_id": "truffleHog:v2",
    "tool_name": "truffleHog",
    "tool_type": "secret-detection",
    "tool_definition": "dockerfile"
    },
    {
    "tool_id": "sonar-scanner-cli:4.4",
    "tool_name": "sonar-scanner-cli",
    "tool_type": "sast",
    "tool_definition": "bash"
    },
    {
    "tool_id": "zap2docker-live:latest",
    "tool_name": "zap2docker-live",
    "tool_type": "dast",
    "tool_definition": "dockerfile"
    }
]
```


401 API key is missing or invalid

404 No Security Tools found!

PoC - sample tests

This document shows more in detail the PoC evaluation results of SecureApps@CI. More specifically, the various test scenarios, referring to distinct analyses and their outcomes.

Secret detection scenario - samples

Secret detection tool - running isolated

```
{
  "urlsWithoutAuthParams": [
    "http://www.google.com/",
    "https://www.google.com/",
    "ftp://localhost:21/",
    "http://www.google.com/test:foobar@/abc"
  ],
  "urlsWithAuthParams": [
    "http://john:doe@www.google.com/",
    "https://john:doe@www.google.com/",
    "ftp://john:doe@localhost:21/",
    "http://john:doe@www.google.com/test:foobar@/abc"
  ],
  "urlsWithEmptyAuthParams": [
    "http://:@www.google.com/",
    "https://:@www.google.com/",
    "ftp://:@localhost:21/",
    "http://:@www.google.com/test:foobar@/abc"
  ]
}
```

Code Snippet 12: Secret detection - credentials inside a file

```
$ trufflehog file:///$PWD/ --json --regex --cleanup --entropy=False | jq -C
{
  "branch": "origin/master",
  "commit": "app2 - sample secrets from repo-supervisor code\n",
  "commitHash": "351bad7199bb427d4aedcb37e33a4c71bd1ea28a",
  "date": "2020-10-09 11:53:57",
  "diff": "@@ -1,20 +0,0 @@\n-{\n-  \"urlsWithoutAuthParams\": ...
  \"ftp://localhost:21/\",\n-    \"http://www.google.com/test:foobar@/abc\"\n-  ],\n-
  \"http://john:doe@www.google.com/\",\n-    \"https://john:doe@www.google.com/\",\n-
  \"http://john:doe@www.google.com/test:foobar@/abc\"\n-  ],\n-  \"urlsWithEmptyAuthParams\":
  \"https://:@www.google.com/\",\n-    \"ftp://:@localhost:21/\",\n-    \"http://:@www...\n",
  "path": "unit/src/filters/entropy.meter/pre.filters/authentication.urls.json",
  "printDiff": "\u001b[93mhttp://john:doe@www.google.com/test:foobar@/abc\"\n\u001b[0m",
  "reason": "Password in URL",
  "stringsFound": [
    "http://john:doe@www.google.com/\",\n",
    "https://john:doe@www.google.com/\",\n",
    "ftp://john:doe@localhost:21/\",\n",
    "http://john:doe@www.google.com/test:foobar@/abc\"\n"
  ]
}
```

Code Snippet 13: Secret detection - running isolated

Secret detection tool - inside solution

```
$ read -s api_key
read> ************************************************************

$ curl --location \
        --header "X-API-KEY: $api_key" \
        --header "Accept: application/json" \
        --header "Content-Type: application/octet-stream" \
        --request POST "https://localhost:5555/ema.rainho/secureapps-ci/v1/analysis/create" \
        --data-binary "@repo-supervisor/Security-Analysis.gitlab-ci.yml" --insecure
```

Code Snippet 14: Secret detection - security analysis pipeline request

```
Running with gitlab-runner 13.6.0 (8fa89735)
  on docker-auto-scale 0277ea0f
> Preparing the "docker+machine" executor
> Preparing environment
> Getting source from Git repository
> Executing "step_script" stage of the job script
    $ apk add --no-cache jq git --update
        (1/6) Installing git (2.26.2-r0)
        (2/6) Installing jq (1.6-r1)
        OK: 31 MiB in 42 packages
    $ python -m pip install --upgrade truffleHog
        Successfully installed GitPython-3.0.6 truffleHog-2.1.11 truffleHogRegexes-0.0.7

    $ trufflehog file:///${PWD}/ --json --regex --entropy=False | tee ${SECRETS_RESULTS} | jq -C
        {
          "branch": "origin/master",
          "commit": "app2 - sample secrets from repo-supervisor code\n",
          "commitHash": "7a29bb736e9b206aabb1aa0f9c5a333a18dbb4e6",
          "date": "2020-10-09 11:53:57",
          "diff": "@@ -1,20 +0,0 @@\n-{\n-  \"urlsWithoutAuthParams\": [\n-    \"http://www.google.com/\",\n-
          "path": "unit/src/filters/entropy.meter/pre.filters/authentication.urls.json",
          "printDiff": "\u001b[93mhttp://john:doe@www.google.com/test:foobar@/abc\"\n\u001b[0m",
          "reason": "Password in URL",
          "stringsFound": [
            "http://john:doe@www.google.com/\",\n",
            "https://john:doe@www.google.com/\",\n",
            "ftp://john:doe@localhost:21/\",\n",
            "http://john:doe@www.google.com/test:foobar@/abc\"\n"
          ]
        }

> Uploading artifacts for failed job
  secrets.json: found 1 matching files and directories
  Uploading artifacts as "archive" to coordinator... ok
> Cleaning up file based variables
ERROR: Job failed: exit code 1
```

Code Snippet 15: Secret detection - inside solution

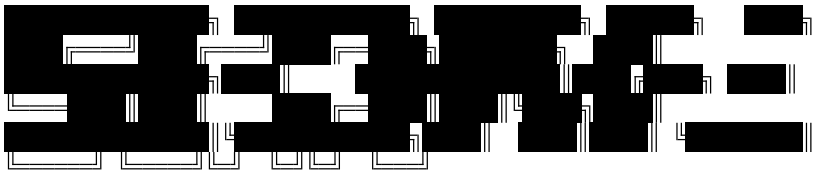
Static analysis scenario - samples

SAST tool - running isolated

	File: requirements.txt
1	Django==1.8.3

Code Snippet 16: SAST - requirements sample

```
$ docker run \
          --rm \
          -v "$PWD:/app" \
          -e "WORKSPACE=$PWD" \
          shiftleft/scan \
          scan \
          --src /app \
          --type python,depscan
```



```
[...] INFO     Scanning /app using plugins ['python', 'depscan']
```

Dependency Scan Results (python)

Id	Package	Used?	Version	Fix Version	Severity	Score
CVE-2019-19844	django	N/A	<1.11.27	1.11.29	CRITICAL	9.8
CVE-2018-7536	django	N/A	>=1.8-<1.8.19	1.11.29	MEDIUM	5.3
CVE-2018-7537	django	N/A	>=1.8-<1.8.19	1.11.29	MEDIUM	5.3
CVE-2017-7233	django	N/A	1.8.3	1.11.29	MEDIUM	6.1
CVE-2017-7234	django	N/A	1.8.3	1.11.29	MEDIUM	6.1
CVE-2016-6186	django	N/A	<=1.8.13	1.11.29	MEDIUM	6.1
CVE-2016-7401	django	N/A	<=1.8.14	1.11.29	HIGH	7.5
CVE-2016-9013	django	N/A	1.8.3	1.11.29	CRITICAL	9.8
CVE-2016-9014	django	N/A	1.8.3	1.11.29	HIGH	8.1

Security Scan Summary

Tool	Critical	High	Medium	Low	Status
Dependency Scan (python)	2	2	5	0	X
Python Source Analyzer	0	0	21	0	X
Python Security Analysis	6	20	34	4	X

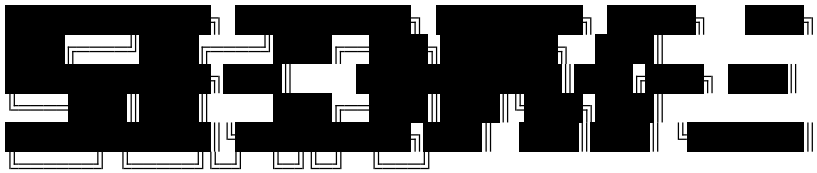
Code Snippet 17: SAST - running isolated

SAST TOOL - INSIDE SOLUTION

```
$ read -s api_key
read> ************************************************************
$ curl --location --header "Content-Type: application/octet-stream"
        --header "X-API-KEY: $api_key" --header "Accept: application/json"
        --request POST "https://localhost:5555/ema.rainho/secureapps-ci/v1/analysis/create" \
        --data-binary "@djangonv/Security-Analysis.gitlab-ci.yml" --insecure
```

Code Snippet 18: SAST - security analysis pipeline request

```
Running with gitlab-runner 13.6.0 (8fa89735)
> Preparing the "docker+machine" executor
> Preparing environment
> Getting source from Git repository
> Executing "step_script" stage of the job script
$ git clone --depth "${GIT_DEPTH}" "${GIT_URL}" "${CI_PROJECT_DIR}/${APP_NAME}"
$ scan --src "${CI_PROJECT_DIR}/${APP_NAME}" --type credscan,depscan,python --out_dir "./reports"
```



```
[.] INFO  Scanning /builds/secureapps-ci/samples/djangonv/django-nv using plugins ['depscan', 'python']
```

Dependency Scan Results (python)

Id	Package	Used?	Version	Fix Version	Severity	Score
CVE-2019-19844	django	N/A	<1.11.27	1.11.29	CRITICAL	9.8
CVE-2018-7536	django	N/A	>=1.8-<1.8.19	1.11.29	MEDIUM	5.3
CVE-2018-7537	django	N/A	>=1.8-<1.8.19	1.11.29	MEDIUM	5.3
CVE-2017-7233	django	N/A	1.8.3	1.11.29	MEDIUM	6.1
CVE-2017-7234	django	N/A	1.8.3	1.11.29	MEDIUM	6.1
CVE-2016-6186	django	N/A	<=1.8.13	1.11.29	MEDIUM	6.1
CVE-2016-7401	django	N/A	<=1.8.14	1.11.29	HIGH	7.5
CVE-2016-9013	django	N/A	1.8.3	1.11.29	CRITICAL	9.8
CVE-2016-9014	django	N/A	1.8.3	1.11.29	HIGH	8.1

Security Scan Summary

Tool	Critical	High	Medium	Low	Status
Dependency Scan (python)	2	2	5	0	X
Python Security Analysis	3	10	17	2	X

```
> Uploading artifacts for failed job
> Cleaning up file-based variables
  ERROR: Job failed: exit code 1
```

Code Snippet 19: SAST - inside solution

Dynamic analysis scenario - samples

DAST tool - running isolated

File: zap-quick-scan.sh

```
#!/usr/bin/env bash

target_url="${TARGET}"
results_file="${REPORT_FILE:-scan_result.txt}"

# run scanner
zap-cli \
    --verbose \
    quick-scan \
    --self-contained \
    --scanners all \
    --spider \
    --ajax-spider \
    --recursive \
    --start-options '-config api.disablekey=true' \
    "${target_url}" \
    -l High | tee "${results_file}"
```

Code Snippet 20: DAST - zap quick-scan script

File: docker-compose.yml

```
version: '3'

services:
  webgoat:
    image: webgoat/webgoat-8.0
    ports:
      - "8088:8080"

  zap2docker:
    image: owasp/zap2docker-weekly
    environment:
      REPORT_FILE: dast_report.txt
      TARGET: http://webgoat:8080/WebGoat
    depends_on:
      - webgoat
    volumes:
      - ./:/zap/wrk/
      - ./zap-quick-scan.sh:/zap/wrk/zap-quick-scan.sh
    entrypoint: /zap/wrk/zap-quick-scan.sh
```

Code Snippet 21: DAST - webgoat and zap deployment

```
$ hostname
laptop.local

$ git clone --depth=1 https://gitlab.com/secureapps-ci/samples/zap-webgoat.git
Cloning into 'zap-webgoat'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 2 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.

$ docker-compose up -d
Creating network "zap-webgoat_default" with the default driver
Creating zap-webgoat_webgoat_1 ... done
Creating zap-webgoat_zap2docker_1 ... done

$ docker logs zap-webgoat_zap2docker_1 -f
[INFO]            Starting ZAP daemon
[DEBUG]           Starting ZAP process with command:
                  ...
                  /zap/zap.sh -daemon \
                              -port 8080 \
                              -config api.disablekey=true.

[DEBUG]           Logging to /zap/zap.log
[DEBUG]           ZAP started successfully.
[INFO]            Running a quick scan for http://webgoat:8080/WebGoat
[DEBUG]           Disabling all current scanners
[DEBUG]           Enabling all scanners
[DEBUG]           Spidering target http://webgoat:8080/WebGoat...
[DEBUG]           Started spider with ID 0...
[DEBUG]           Spider progress %: 0
[DEBUG]           Spider #0 completed
[DEBUG]           AJAX Spidering target http://webgoat:8080/WebGoat...
[DEBUG]           AJAX Spider: running
[DEBUG]           AJAX Spider completed
[DEBUG]           Scanning target http://webgoat:8080/WebGoat...
[DEBUG]           Started scan with ID 0...
[DEBUG]           Scan progress %: 0
[DEBUG]           Scan progress %: 60
[DEBUG]           Scan progress %: 100
[DEBUG]           Scan #0 completed
[INFO]            Issues found: 4
+-----------------------+------+--------+----------------------------------------+
| Alert                 | Risk | CWE ID | URL                                    |
+=======================+======+========+========================================+
| Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/register.mvc |
+-----------------------+------+--------+----------------------------------------+
| Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/login      |
+-----------------------+------+--------+----------------------------------------+
| Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/login?error |
+-----------------------+------+--------+----------------------------------------+
| Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/registration |
+-----------------------+------+--------+----------------------------------------+
[INFO]            Shutting down ZAP daemon
[DEBUG]           Shutting down ZAP.
[DEBUG]           ZAP shutdown successfully.
```

Code Snippet 22: DAST - running isolated at localhost

DAST TOOL - INSIDE SOLUTION

```
$ read -s api_key
read> ************************************************************

$ curl --location --header "Content-Type: application/octet-stream" \
        --header "X-API-KEY: $api_key" --header "Accept: application/json" \
        --request POST "https://localhost:5555/ema.rainho/secureapps-ci/v1/analysis/create" \
        --data-binary "@zap-webgoat/Security-Analysis.gitlab-ci.yml" --insecure
```

Code Snippet 23: DAST - security analysis pipeline request

```
> Running with gitlab-runner 13.6.0 (8fa89735)
> Preparing the "docker+machine" executor
> Getting source from Git repository
> Executing "step_script" stage of the job script
    $ docker-compose up -d webgoat webwolf
        Creating zap-webgoat_webgoat_1
    $ docker-compose run -d -e ENV=dev \
                         -e REPORT_FILE="$DAST_REPORT_FILE" \
                         -e TARGET="$TEST_TARGET" zap2docker
        Pulling zap2docker (owasp/zap2docker-weekly:latest)...
        zapwebgoat_zap2docker_run_1

    $ docker logs zapwebgoat_zap2docker_run_1 -f
        [DEBUG]           Starting ZAP process with command:
                          /zap/zap.sh -daemon \
                                      -port 8080 \
                                      -config api.disablekey=true.
        [DEBUG]           ZAP started successfully.
        [INFO]            Running a quick scan for http://webgoat:8088/WebGoat
        [DEBUG]           Enabling all scanners
        [DEBUG]           Spider #0 completed
        [DEBUG]           AJAX Spider completed
        [DEBUG]           Scanning target http://webgoat:8088/WebGoat...
        [DEBUG]           Scan #0 completed
        [INFO]            Issues found: 4
        +-----------------------+------+--------+-----------------------------------------+
        | Alert                 | Risk | CWE ID | URL                                     |
        +=======================+======+========+=========================================+
        | Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/register.mvc |
        +-----------------------+------+--------+-----------------------------------------+
        | Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/login        |
        +-----------------------+------+--------+-----------------------------------------+
        | Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/login?error  |
        +-----------------------+------+--------+-----------------------------------------+
        | Anti-CSRF Tokens Check | High |    352 | http://webgoat:8080/WebGoat/registration |
        +-----------------------+------+--------+-----------------------------------------+

        [DEBUG]           ZAP shutdown successfully.
        Uploading artifacts...
        ERROR: Job failed: exit code 1
```

Code Snippet 24: DAST pipeline - inside solution

External analysis scenario - samples

External tool - running isolated

File: sonar-project.properties

```
0  sonar.language=java
1  sonar.java.source=1.8
2  sonar.sources=src/main/java
3  sonar.java.binaries=target/classes
4  sonar.projectKey=secureapps-ci_hello-shiftleft
5  sonar.organization=secureapps-ci
6
7  # This is the name and version displayed in the SonarCloud UI.
8  sonar.projectName=hello-shiftleft
9  sonar.projectVersion=1.0
```

Code Snippet 25: External analysis - sonar-project.properties file

```
<properties>
    <java.version>1.8</java.version>
    <jackson.mapper.version>1.5.6</jackson.mapper.version>
    <sonar.projectKey>secureapps-ci_hello-shiftleft</sonar.projectKey>
    <sonar.organization>secureapps-ci</sonar.organization>
</properties>
```

Code Snippet 26: External analysis - pom.xml file

```
$  mvn clean package

[INFO] Scanning for projects...
[INFO] Building hello-shiftleft 0.0.1
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ hello-shiftleft ...
[INFO] Deleting $HOME/secureapps-ci/samples/hello-shiftleft/target
[INFO] --- maven-compiler-plugin:3.6.1:compile (default-compile) @ hello-shiftleft ...
[INFO] Compiling source files to $HOME/secureapps-ci/samples/hello-shiftleft/target/classes ...
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ hello-shiftleft ...
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory $HOME/secureapps-ci/samples/...
[INFO] --- maven-compiler-plugin:3.6.1:testCompile (default-testCompile)...
[INFO] No sources to compile
[INFO] --- maven-surefire-plugin:2.18.1:test (default-test) @ hello-shift...
[INFO] No tests to run.
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ hello-shiftleft ...
[INFO] Building jar: $HOME/secureapps-ci/samples/hello-shiftleft/target/hello-shiftleft-0.0.1.jar
[INFO] --- spring-boot-maven-plugin:1.5.1.RELEASE:repackage (default) @ hello-shiftleft ...
[INFO] BUILD SUCCESS
[INFO] Total time:  6.356 s
[INFO] Finished at: 2020-12-27T18:38:41Z
```

Code Snippet 27: External analysis - running build locally

```
$ read -s SONAR_TOKEN
read> ****************************************

$ read SONAR_HOST_URL
read> https://sonarcloud.io

$ sonar-scanner -Dsonar.login="$SONAR_TOKEN" -Dsonar.host.url="$SONAR_HOST_URL"

    INFO: Scanner configuration file: /src/sonar-scanner/4.5.0.2216/libexec/conf/sonar-scanner.properties
    INFO: SonarScanner 4.5.0.2216
    INFO: Java 13.0.2 Oracle Corporation (64-bit)
    INFO: Mac OS X 10.15.7 x86_64
    INFO: Load global settings
    INFO: Server id: 1BD809FA-AWHW8ct9-T_TB3XqouNu
    INFO: Load/download plugins
    INFO: Load project settings for component key: 'secureapps-ci_hello-shiftleft' (done) | time=112ms
    INFO: Process project properties
    INFO: Execute project builders
    INFO: Project key: secureapps-ci_hello-shiftleft
    INFO: Working dir: ~/secureapps-ci/samples/hello-shiftleft/.scannerwork
    INFO: Read 78 type definitions
    INFO: Reading UCFGs from: ~/secureapps-ci/samples/hello-shiftleft/.scannerwork/ucfg2/java
    INFO: 19:03:19.431414 Building Runtime Type propagation graph
    INFO: 19:03:19.441532 Running Tarjan on 161 nodes
    INFO: 19:03:19.442918 Tarjan found 161 components
    INFO: 19:03:19.44631 Variable type analysis: done
    INFO: Analyzing 113 ucfgs to detect vulnerabilities.
    INFO: All rules entrypoints : 0 Retained UCFGs : 0
    INFO: Sensor JavaSecuritySensor [security] (done) | time=772ms
    INFO: Reading type hierarchy from: ~/secureapps-ci/samples/hello-shiftleft/ucfg_cs2
    INFO: ------------- Run sensors on project
    INFO: Sensor Zero Coverage Sensor
    INFO: Sensor Zero Coverage Sensor (done) | time=29ms
    INFO: Sensor Java CPD Block Indexer
    INFO: Sensor Java CPD Block Indexer (done) | time=260ms
    INFO: CPD Executor 7 files had no CPD blocks
    INFO: CPD Executor Calculating CPD for 14 files
    INFO: CPD Executor CPD calculation finished (done) | time=33ms
    INFO: Analysis report generated in 508ms, dir size=267 KB
    INFO: Analysis report compressed in 355ms, zip size=93 KB
    INFO: Analysis report uploaded in 1945ms
    INFO: ANALYSIS SUCCESSFUL, ... results at:
    https://sonarcloud.io/dashboard?id=secureapps-ci_hello-shiftleft
    report processing at https://sonarcloud.io/api/ce/task?id=AXallXBM0CVG157Sg715
    INFO: Analysis total time: 27.776 s
    INFO: ------------------------------------------------------------------------
    INFO: EXECUTION SUCCESS
    INFO: ------------------------------------------------------------------------
    INFO: Total time: 17:31.817s
    INFO: Final Memory: 36M/134M
    INFO: ------------------------------------------------------------------------
```

Code Snippet 28: External analysis - running sonar scanner locally

External tool - inside solution

```
$ read -s api_key
read> ************************************************************

$ curl --location --header "Content-Type: application/octet-stream" \
        --header "Accept: application/json" --header "X-API-KEY: $api_key" \
        --request POST "https://localhost:5555/ema.rainho/secureapps-ci/v1/analysis/create" \
        --data-binary "@hello-shiftleft/Security-Analysis.gitlab-ci.yml" --insecure
```

Code Snippet 29: External tool - security analysis pipeline request

```
> Running with gitlab-runner 13.7.0-rc1 (98e2e32d)
> Resolving secrets
> Preparing the "docker+machine" executor
> Preparing environment
> Getting source from Git repository
> Executing "step_script" stage of the job script
    $ mvn clean package
    [INFO] Scanning for projects...
    Downloading from central: \
    https://repo.maven.apache.org/maven2/org/.../spring-boot-starter-parent-1.5.1.RELEASE.pom
    Progress (1): 2.7/7.4 kB
    Progress (1): 5.5/7.4 kB
    Progress (1): 7.4 kB
    ...
    [INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ hello-shiftleft ---
    [INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ hello-shiftleft ---
    [INFO] --- maven-compiler-plugin:3.6.1:compile (default-compile) @ hello-shiftleft ---
    [INFO] Changes detected - recompiling the module!
    [INFO] Compiling 21 source files to /builds/../../hello-shiftleft/target/classes
    [INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ hello-shiftleft ---
    [INFO] Using 'UTF-8' encoding to copy filtered resources.
    [INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ hello-shiftleft ---
    [INFO] Building jar: /builds/../../hello-shiftleft/target/hello-shiftleft-0.0.1.jar
    [INFO] --- spring-boot-maven-plugin:1.5.1.RELEASE:repackage (default) @ hello-shiftleft ---

> Uploading artifacts for successful job
    Uploading artifacts...
    target/: found 53 matching files and directories
    Uploading artifacts as "archive" to coordinator... ok
> Cleaning up file based variables
> Job succeeded
```

Code Snippet 30: External tool pipeline - running build inside Solution

```
> Running with gitlab-runner 13.7.0-rc1 (98e2e32d)
> Preparing the "docker+machine" executor
> Preparing environment
> Getting source from Git repository

    $ sonar-scanner
    INFO: Scanner configuration file: /opt/sonar-scanner/conf/sonar-scanner.properties
    INFO: SonarScanner 4.5.0.2216
    INFO: Java 11.0.6 AdoptOpenJDK (64-bit)
    INFO: Linux 4.19.78-coreos amd64
    INFO: Quality profile for java: Sonar way
    INFO: ------------- Run sensors on module secureapps-ci_hello-shiftleft
    INFO: Configured Java source version (sonar.java.source): 8
    INFO: JavaClasspath initialization (done) | time=17ms
    INFO: Java Main Files AST scan
    INFO: Analyzing 113 ucfgs to detect vulnerabilities.
    INFO: All rules entrypoints : 0 Retained UCFGs : 0
    INFO: Sensor JavaSecuritySensor [security] (done) | time=981ms
    INFO: Reading type hierarchy from: /builds/../../hello-shiftleft/ucfg_cs2
    INFO: Analysis report uploaded in 435ms
    INFO: ANALYSIS SUCCESSFUL, you can find the results at:
    https://sonarcloud.io/dashboard?id=secureapps-ci_hello-shiftleft&branch=master
    report processing at https://sonarcloud.io/api/ce/task?id=AXZ1XF1dIcbAprWxlP-B
    INFO: Analysis total time: 20.323 s
    INFO: ------------------------------------------------------------------------
    INFO: EXECUTION SUCCESS
    INFO: ------------------------------------------------------------------------
    INFO: Total time: 37.614s
    INFO: Final Memory: 35M/98M
> Cleaning up file-based variables
> Job succeeded
```

Code Snippet 31: External tool pipeline - running sonar scanner inside Solution

Integration analysis scenario - samples

Diversity of tools - inside solution

```
$ read -s api_key
read> ************************************************************

$ curl --location \
       --header "X-API-KEY: $api_key" \
       --header "Accept: application/json" \
       --header "Content-Type: application/octet-stream" \
       --request POST "https://localhost:5555/ema.rainho/secureapps-ci/v1/analysis/create" \
       --data-binary "@hello-shiftleft/Security-Analysis.gitlab-ci.yml" --insecure
```

Code Snippet 32: Diversity of tools - security analysis pipeline request

Solution achieved requirements

Deployment

```
FROM python:3.9-alpine AS build-env
RUN mkdir -p /usr/src/app/ && \
    addgroup --gid 9999 appgroup && \
    adduser --uid 9999 -D -G appgroup -h /usr/src/app appuser && \
    apk add --no-cache --update gcc musl-dev libffi-dev openssl-dev && \
    chown appuser:appgroup -R /usr/src/app/ && \
    python3 -m pip install --no-cache-dir --upgrade pip setuptools pip-autoremove
WORKDIR /usr/src/app
USER appuser
COPY requirements.txt /tmp
COPY ./swagger_server/ /usr/src/app/swagger_server/
RUN python3 -m pip install --user --no-cache-dir safety && \
    python3 -m pip install --user --no-cache-dir --upgrade -r /tmp/requirements.txt && \
    if ! python3 -m pip freeze | safety check --stdin; then exit; fi && \
    pip-autoremove -y safety

FROM python:3.9-alpine
ENV PATH="/usr/src/app/.local/bin:$PATH"
HEALTHCHECK --interval=5m --timeout=3s \
  CMD wget -nv -t1 --spider --no-check-certificate \
      https://localhost:5555/ema.rainho/secureapps-ci/v1/health || exit 1
RUN apk -U upgrade && \
    mkdir -p /usr/src/app/ && \
    addgroup --gid 9999 appgroup && \
    adduser --uid 9999 -D -G appgroup -h /usr/src/app appuser && \
    python3 -m pip uninstall -y pip
COPY --chown=appuser:appgroup \
     --from=build-env \
     /usr/src/app/swagger_server /usr/src/app/swagger_server
COPY --chown=appuser:appgroup \
     --from=build-env \
     /usr/src/app/.local /usr/src/app/.local
WORKDIR /usr/src/app
USER appuser
EXPOSE 5555
ENTRYPOINT ["python3"]
CMD ["-m", "swagger_server"]
```

Code Snippet 33: Multistage Dockerfile sample

Definition analysis

```
variables:
  APP_NAME: "HelloShiftLeft"
  GROUP_NAME: "vulnerable-apps"
  GIT_URL: "https://github.com/ShiftLeftSecurity/HelloShiftLeft.git"
  BRANCH_NAME: "master"
  GIT_DEPTH: '1'
  ENV: "dev"

stages:
  - secret_detection
  - static_analysis

secrets:
  stage: secret_detection
  variables:
    SECRETS_RESULTS: 'secrets.json'
  script:
    - trufflehog ${CI_PROJECT_DIR}/ \
                 --json \
                 --regex \
                 --entropy=False \
                 | tee ${CI_PROJECT_DIR}/${SECRETS_RESULTS} | jq -C
  artifacts:
    paths: ["${SECRETS_RESULTS}"]
    when: always
  allow_failure: false

sast:
  stage: static_analysis
  needs: ["secrets"]
  image:
    name: shiftleft/sast-scan
  script:
    - scan --src ${CI_PROJECT_DIR} \
           --type depscan,python \
           --out_dir ${CI_PROJECT_DIR}/reports
  rules:
    - when: always
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - $CI_PROJECT_DIR/reports/
    when: always
  allow_failure: true
```

Code Snippet 34: Definition analysis form