This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No. 101069732

aerOS DevPrivSecOps cookbook

Copyright

The aerOS consortium consists of the following 27 partners::

UNIVERSITAT POLITÈCNICA DE VALÈNCIA	ES
NATIONAL CENTER FOR SCIENTIFIC RESEARCH "DEMOKRITOS"	EL
ASOCIACION DE EMPRESAS TECNOLOGICAS INNOVALIA	ES
TTCONTROL GMBH	AT
TTTECH COMPUTERTECHNIK AG (*third linked party*)	AT
SIEMENS AKTIENGESELLSCHAFT	DE
FIWARE FOUNDATION EV	DE
TELEFONICA INVESTIGACION Y DESARROLLO SA	ES
ORGANISMOS TILEPIKOINONION TIS ELLADOS OTE AE - HELLENIC TELECOMMUNICATIONS ORGANIZATION SA	EL
EIGHT BELLS LTD	CY
INQBIT INNOVATIONS SRL	RO
FOGUS INNOVATIONS & SERVICES P.C.	EL
L.M. ERICSSON LIMITED	IE
SYSTEMS RESEARCH INSTITUTE OF THE POLISH ACADEMY OF SCIENCES IBS PAN	PL
ICTFICIAL OY	FI
INFOLYSIS P.C.	EL
PRODEVELOP SL	ES
EUROGATE CONTAINER TERMINAL LIMASSOL LIMITED	CY
TECHNOLOGIKO PANEPISTIMIO KYPROU	CY
DS TECH SRL	IT
GRUPO S 21SEC GESTION SA	ES
JOHN DEERE GMBH & CO. KG*JD	DE
CLOUDFERRO S.A.	PL
ELECTRUM SP ZOO	PL
POLITECNICO DI MILANO	IT
MADE SCARL	IT
NAVARRA DE SERVICIOS Y TECNOLOGIAS SA	ES
SWITZERLAND INNOVATION PARK BIEL/BIENNE AG	CH

Disclaimer

Authors

Name	Partner	e-mail
Prof. Carlos E. Palau Ignacio Lacalle Úbeda Rafael Vaño Raúl San Julián	P01 UPV	cpalau@upv.es iglaub@upv.es ravagar2@upv.es rausanga@upv.es
Korbinian Pfab	P05 SIEMENS	korbinian.pfab@siemens.com
Christos Xenakis Giannis Chouchoulis Giannis Makropodis Giorgos Petihakis	P10 IQB	chris@inqbit.io giannis.makropodis@inqbit.io giannis.chouchoulis@inqbit.io giorgos.petihakis@inqbit.io
Katarzyna Wasielewska-Michniewska Maria Ganzha Marcin Paprzycki	P13 IBSPAN	katarzyna.wasielewska@ibspan.waw.pl maria.ganzha@ibspan.waw.pl marcin.paprzycki@ibspan.waw.pl
Oscar Lopez Jon Egaña Rafael Borne Ramiro Torres	P20 S21Sec	jon.egana@thalesgroup.com oscar.lopez-perez@thalesgroup.com rafael.borne@thalesgroup.com ramiro.torres@thalesgroup.com
Daniel Cobo Borrega	P26 NASERTIC	dcobobor@nasertic.es

History

Date	Version	Change
18/03/2024	0.1	TOC definition
29/05/2024	0.2	Partners final contribution
31705/2024	1.0	Final version

Executive Summary

This document presents the DevPrivSecOps methodology being designed within the aerOS project. This methodology will allow the internal developers and also the global developers landscape to bring the right practices in the lifecycle of the software developed during the project. Especially, and considering that the DevOps methodology is known and applied by most of the developers, efforts have been directed to the implementation of security and privacy within the DevOps methodology.

In response to the growing emphasis on privacy considerations, aerOS proposes the integration of privacy controls into the existing DevSecOps methodology, leading to the development of DevPrivSecOps. This novel approach aims to advance beyond the current industry standard and will incorporate privacy requirements, enabling aerOS developers to design software that is both secure and privacy-conscious from the beginning of the process.

This document presents an implementation guide for the DevPrivSecOps methodology designed within the aerOS project. By following the steps defined in the document, developers will be able to implement the methodology and validate that the developed code is secure and privacy compliant by design.

To this end, a series of tests have been designed and implemented that are executed automatically and manually to analyse and detect any security or privacy problems that the developed code may have. To implement this methodology, some open-source tools have been selected and some of them have been modified according to the needs of the project.

This methodology is intended to teach, not only aerOS developers, but also the entire software development landscape how to implement it.

.

Table of contents

List of tables

List of figures

List of acronyms

Acronym	Explanation
AST	Abstract Syntax Trees
API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
CPD	Continuous Planning Design and development
DAST	Dynamic Application Security Testing
DevOps	Development and Operations
DevPrivSecOps	Development, Privacy, Security and Operations
DevSecOps	Development, Security and Operations
GDPR	General Data Protection Regulation
IDE	Integrated Development Environment
IT	Information technology
JSON	JavaScript Object Notation
OWASP	Open Web Application Security Project
SAST	Static application software testing
SCA	Software Composition Analysis
SSL	Secure Sockets Layer
SW	Software
TLS	Transport Layer Security
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	Yet Another Markup Language

Table 1 List of acronyms

1. Introduction

Due to the advances in the different types of attacks and the number of attacks that are carried out daily on systems exposed on the internet, we are currently faced with the need to generate secure and privacy aware code by design.

Security by design principles should be applied during the design phase of a product's development life cycle to drastically reduce the number of exploitable failures before they are introduced to the market for widespread use or consumption [1]. In addition, security must be analysed at every stage of the code development life cycle to achieve a final product free of vulnerabilities and security problems. However, this is not enough, as new attacks (so-called zero-day attacks) are discovered almost every day [2], which means that software components need constant maintenance to be able to correct these newly detected vulnerabilities.

In recent years, the need to protect the personal data that applications use has increased and to this end, several regulations related to data protection have been published. Specifically, in Europe the main regulation for data protection is the General Data Protection Regulation (GDPR), published in 2016 [3]. It is now widely regarded as a privacy law not only for the EU [4], but for the whole world.

These regulations must be applied to all environments where data is used. Even if during the development of the SW developers do not use any private data, they need to ensure that once the SW is deployed, the operation complies with the above-mentioned regulations. This adds an important requirement for software developers.

In the same way as security, privacy analysis should be carried out in all phases of software development lifecycle. It should be mentioned that a more thorough privacy analysis is performed once the software is running with business data, but as mentioned above, this is not sufficient. This need leads to the evolution of the software development methodology from DevSecOps to DevPrivSecOps. The aim of this is to increase the security and privacy knowledge of developers, testers, and operations staff and increase the partnership of privacy and security experts.

The methodology presented in this document has been designed with the intention of ensuring that the developments carried out within the project are secure and have privacy by design. In addition, a procedure is presented to ensure that the design, development and deployment of the software is done in a secure and privacy-compliant manner. Finally, the toolset that is used in aerOS and how these should be used to achieve this objective is presented.

The main objective of this document is to guide developers and staff in charge of deploying aerOS components to configure and use the different tools selected to implement the DevPrivSecOps methodology designed in the project. This document has been enriched with examples to facilitate the use of each of the tools.

2. aerOS DevPrivSecOps methodology

The term DevOps is a combination of the two terms development and operations, representing a collaborative approach to the tasks that are performed by development and IT operations teams. DevOps is a collection of several tools and practices that help automate and integrate the processes between IT professionals and software. It further focuses on team collaboration, team empowerment, cross-team communication, and technology automation. DevOps can co-exist with IT service management frameworks, Agile software development, project management directives, and other IT infrastructures. DevOps can be simply defined as *a methodology that helps optimizing the process of software development and operation* [5].

DevSecOps means thinking about application and infrastructure security from the beginning and embedding DevOps with security controls providing continuous security assurance [6]. DevSecOps is a natural extension of DevOps to include security-by-design and continuous security testing by automating some security controls in the DevOps workflow.

By adding tests to the DevSecOps methodology to ensure that privacy can be analysed in all phases of the process, we get the DevPrivSecOps methodology. This methodology aims to generate secure and privacy-aware code from the design phase.

The Figure 1 presents the DevPrivSecOps methodology approach for aerOS (the Figure 2 represents the methodology with the selected tools), where a sequence of steps describes how it should be implemented from design through development to deployment and monitoring. The steps to be followed are:

1. Continuous planning and tracking for development testing and deployment activities: security and privacy threat modelling.
2. Code using IDE tools and SAST tools plugins integrated into IDE.
3. Commit code in git source version control repository.
4. Integration automation process pulls the source and build the application.
5. Integration automation may run SCA for dependency check and launch SAST tools.
6. Integration automation may run Acceptance, Smoke, Load and Performance Testing, with Unit and Integration Testing for Integration and Security test execution.
7. Integration automation launches Orchestration Platform and Configuration Management for configuration and build deployment at pre-production server.
8. Integration automation may run Acceptance, Smoke, Load and Performance Testing for Application Acceptance and Security and Privacy test execution.
9. Integration automation launches DAST tools for vulnerability scanner, pentesting and exploit tests.
10. Build automation tool uploads tested package to artifact repository.
11. Wait for approval for Production Deployment.
12. Integration automation launches Orchestration Platform for production server configuration and build deployment.
13. Configuration management download tested package from Artifact repository and deploys to production.
14. Integration automation may launch Acceptance, Smoke, Load and Performance Testing and DAST tools for vulnerability scanner, pentesting, privacy test and exploit test at production server.
15. Production server goes live with updated application and continuous operation.
16. Continuous operation with logging, analysis, visualization, and notification tools. Continuous Monitoring: Security Information, Privacy Information, Event Management and DAST tools
17. Continuous feedback through all the stages development, build, integration, testing and deployment tools at different stages of the workflow.

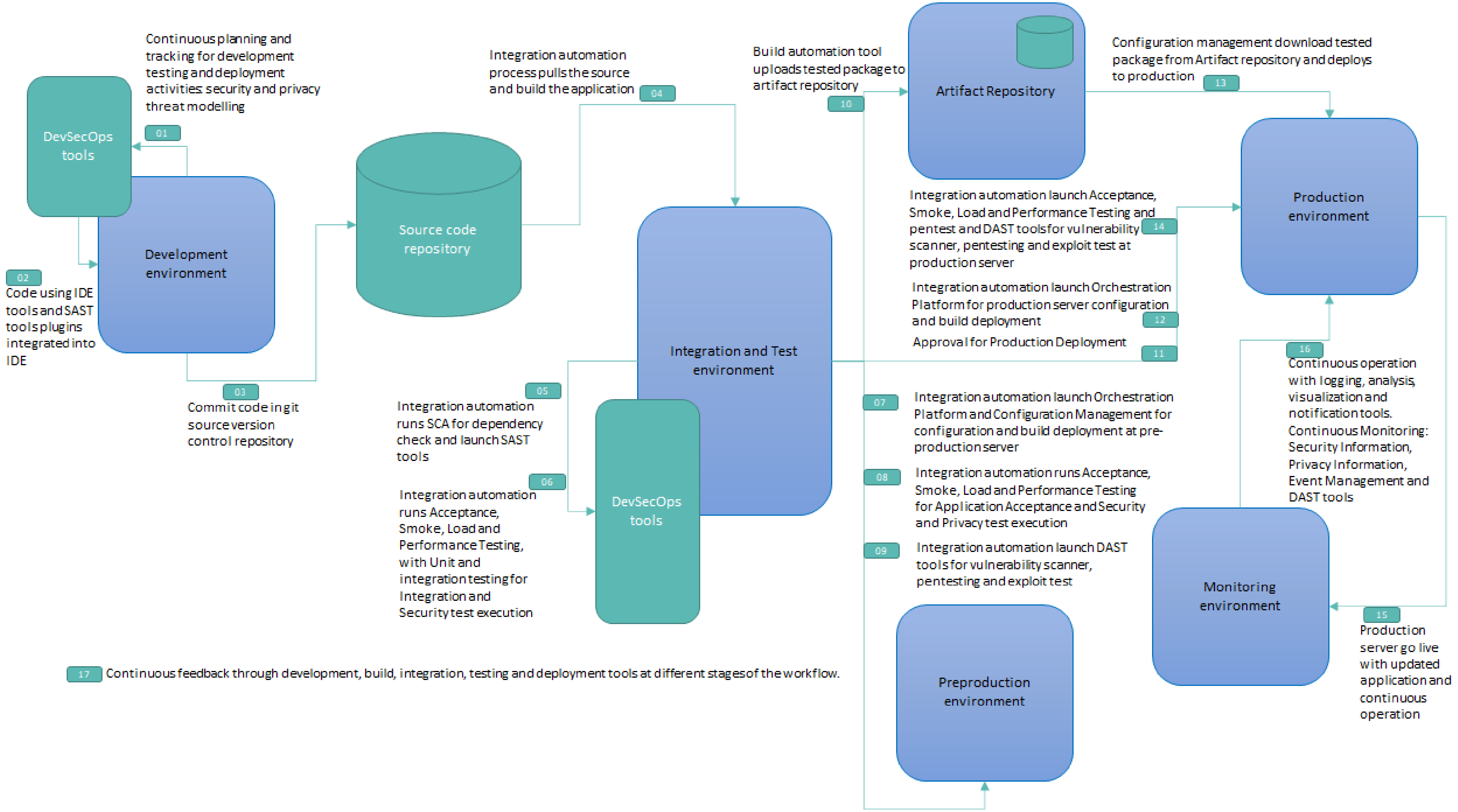

Figure 1 aerOS DevPrivSecOps methodology.

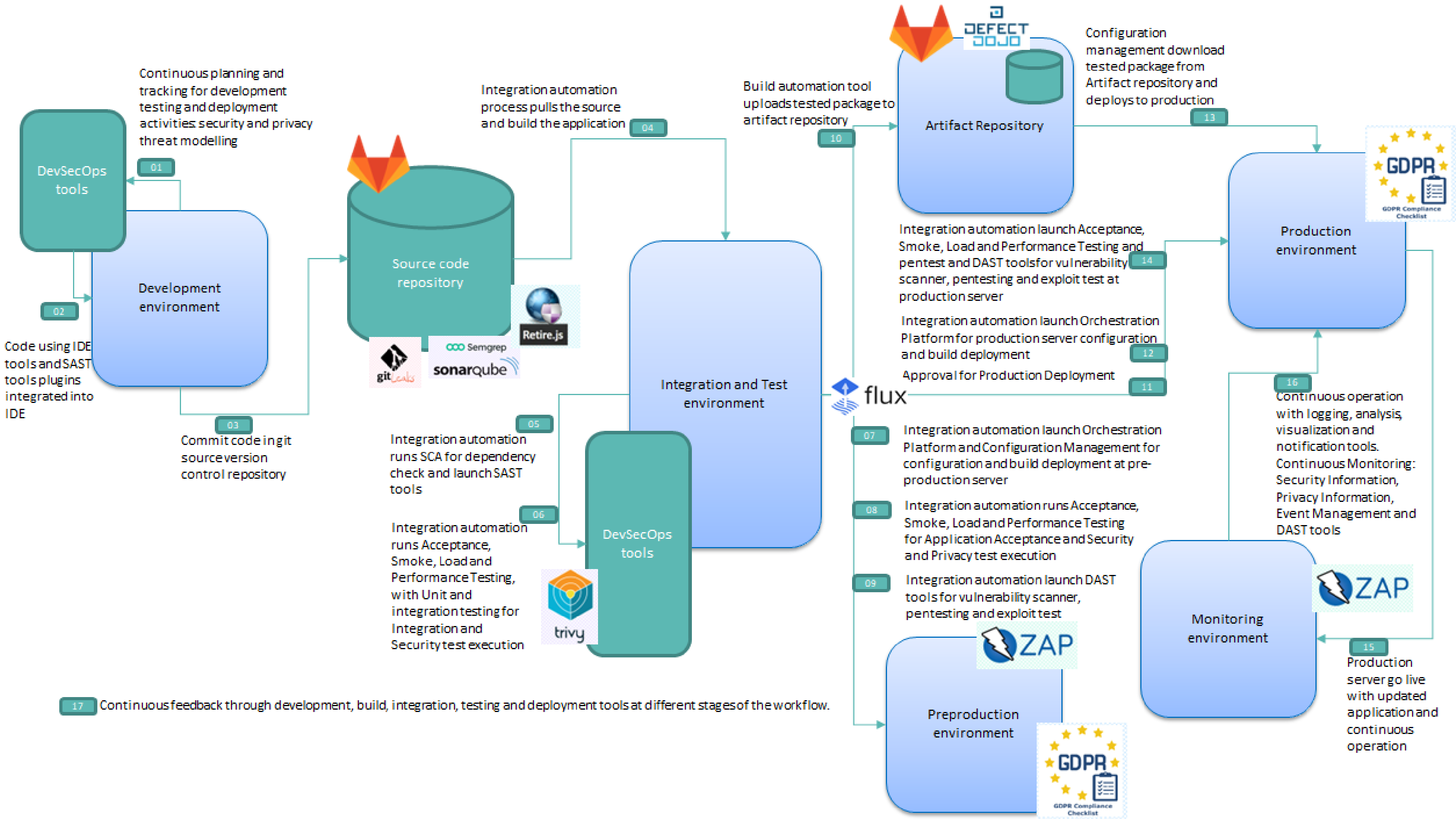

Figure 2 aerOS DevPrivSecOps methodology with tools.

3. DevPrivSecOps toolset implementation

This section presents the different tools used to implement the DevPrivSecOps methodology.

Figure 3 shows the CI/CD pipeline that has been designed to implement the methodology and the tools that have been used in each step of this pipeline: GitLab [7], GitLeaks [8], Semgrep [9], SonarQube [10], Retire.js [11], Trivy [12], Flux CD [13], ZAP proxy [14] and GDPR checklist.

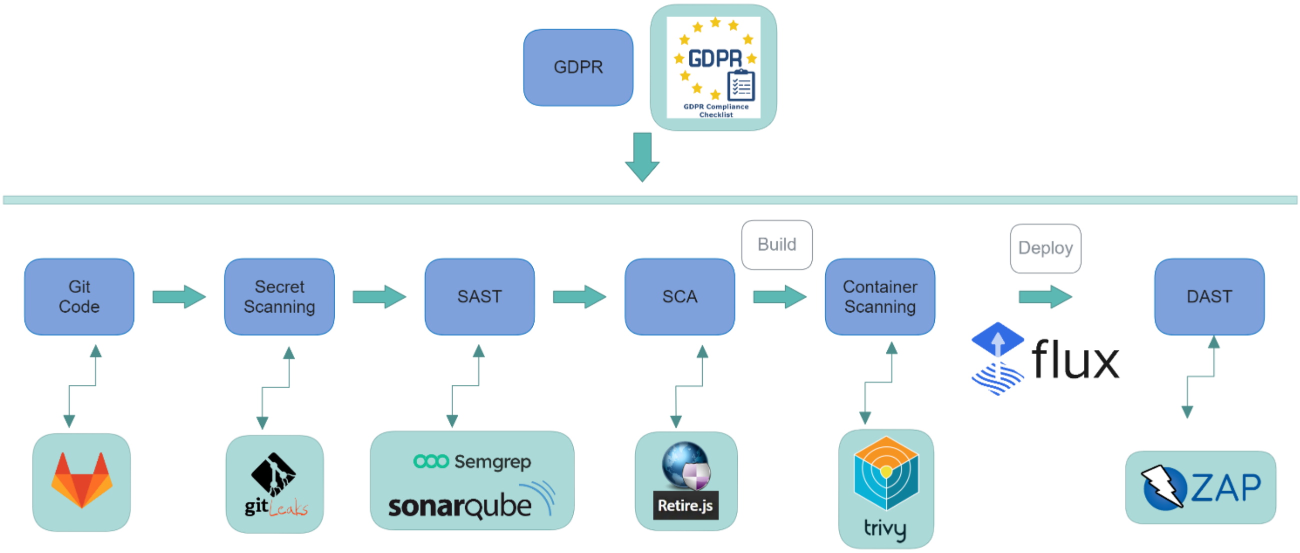

Figure 3 aerOS DevPrivSecOps implementation CI/CD pipeline.

Gitlab, in addition to the software version control repository, is in charge of launching the different steps through the CI/CD pipeline configuration. The steps taken to implement the DevPrivSecOps methodology are as follows:

- **Secret Scanning**: GitLeaks has been used to scan if secrets have been included in the code. This tool allows us to analyse the code for privacy compliance, ensuring that user data is not accessible to people who have access to the repository.
- **SAST**: Static testing is performed once the software has been uploaded to the change control repository. These tests allow to analyse the vulnerabilities and security problems that the code may have. Semgrep and SonarQube will be used for this purpose in the project.
- **SCA**: Retire.js has been implemented for the implementation of Software Composition Analysis tests in the project. This test analyses the dependencies of the code, looking for any kind of security risks such as vulnerability dependencies.
- **Container Scanning:** Trivy has been implemented in order to be able to analyse the traceability of containerised components. Since most of the components of the project are containerised, it was decided to implement this test which complements the others.
- **Continuous Deployment:** In aerOS, it has been decided to use Flux CD as the continuous deployment component of the software developed in the project. This SW is dedicated to monitoring the changes in the master branch of the repositories, and when it detects any type of change in these, it updates the component in the production environment of the pilots.
- **DAST:** ZAP has been chosen to perform dynamic tests on the components deployed in the pilot environment. This tool enables security tests to be carried out on the components deployed, thus detecting security flaws in them.
- **GDPR:** In order to analyse GDPR compliance in software development, a check-list has been used in the project to analyse the status of compliance in the different phases.

In the following sections, the tools mentioned, and others that were used to implement the DevPrivSecOps methodology will be analysed in more detail and guidelines for the use of these tools will be provided in the form of examples.

3.1. Collaboration and communication tool: Mattermost

Mattermost is an open-source online chat service mainly focused on companies and organisations. This service, which is hosted on the UPV partner's servers and is publicly accessible through the URL https://mattermost.aeros-project.eu/, allows to create communication channels according to a topic, follow messages through threads or start private conversations with other users. The main purpose of Mattermost in the aerOS project is to offer a robust, secure and highly available communication system between all the partners involved.

Currently, there are more than 30 channels on the aerOS Mattermost organised by theme. Mainly there are channels focused on work packages and tasks. Work package channels are used to share general information about the work package, such as changes in meeting dates, reminders, deliverables, etc. Task channels focus on more technical and developmental aspects between the different partners involved to streamline the exchange of ideas, concepts, etc., and even documentation such as source code, figures, diagrams or schematics. These channels also speed up the programming of the different aerOS components by allowing the teams to maintain direct and uninterrupted communication to improve the integration of the different modules that compose the meta-operating system.

Also, participants in the chats can share code files, and even source code within the messages in a user-friendly way. In addition, these channels have also been used to discuss doubts about the integration of the different modules that compose the main elements of aerOS.

Another Mattermost feature is the possibility to use threads within conversations. This allows not to divert the main attention of the channel and to focus the threads on more specific topics or technical aspects about a specific question, component or technology. In this way, the main conversation of the channel can be followed in a better way, avoiding the saturation of messages and notifications.

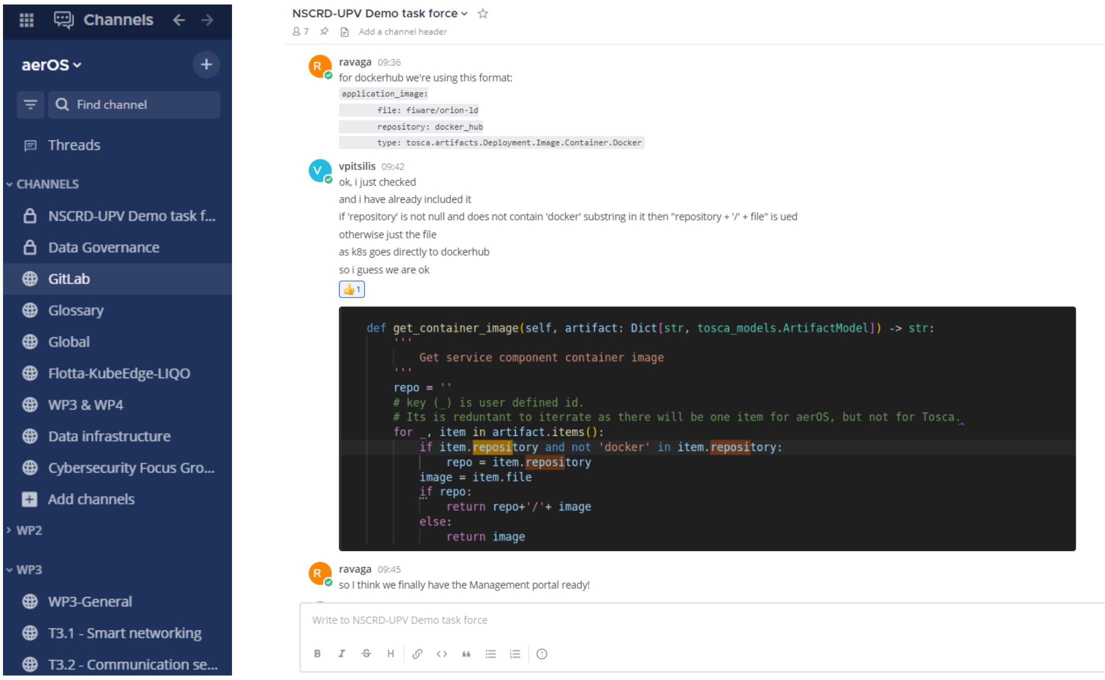

Figure 4 Channels organization and example of a channel for developers.

Finally, it is possible to create channels on demand, in case a partner or group of project partners need a specific communication channel for a specific topic or technology that does not exist before in Mattermost. The channels that are created can be public (any registered user can participate in them) or private (only invited participants can access them). This way it is possible to have, for instance, private groups for developers to accelerate pending integration tasks among software development teams.

3.2. Source version control and CPD: GitLab

Gitlab has been selected as the best solution for managing source code repositories in aerOS, which also includes additional functionalities such as CI/CD support, container images registry and installation packages registry, among others. Therefore, a dockerized instance of the Community Edition version of GitLab, which is being constantly being updated to avoid security issues, was installed in the private infrastructure of the UPV partner. This Gitlab repository can be accessed through the public URL https://gitlab.aeros-project.eu/, as well as its API, and the container registry of aerOS (registry.gitlab.aeros-project.eu).

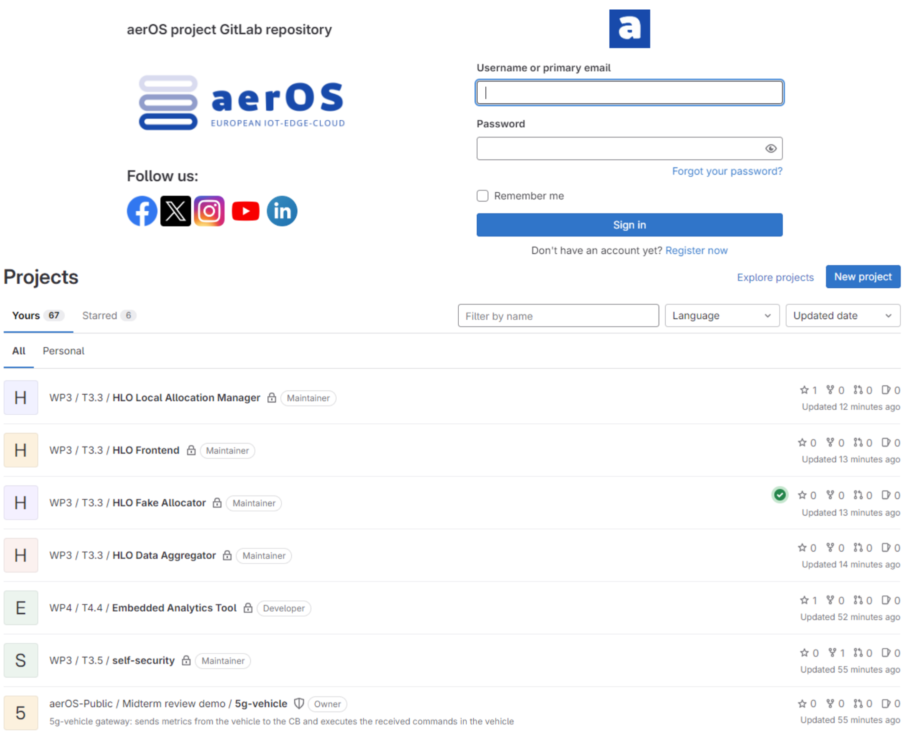

Figure 5 aerOS project private Gitlab.

In a project like aerOS with a large number of partners, which is translated into more than 100 registered users in Gitlab, it was identified the need of defining a clear and transparent strategy to organize the code, projects or repositories in Gitlab and to manage the access permissions. Therefore, it was decided to create a group for each task to include all the software developed within the scope of that task. Task leaders are assigned with a *Maintainer* role (higher permissions), whereas task participants are assigned with a *Developer* role. Moreover, a private group has been created for each partner to speed up internal developments and content sharing without the need for an additional tool.

3.2.1. CI/CD

Gitlab provides full support to incorporate CI/CD pipelines in the software development and delivery process. Specifically, in Gitlab the CI/CD process is performed by a runner, which runs a series of jobs listed in a YAML file (the *.gitlab-ci.yaml* file, which is located in the root path of a Gitlab code repository) and reports their final results to Gitlab in order to show them to the final user in a user-friendly dashboard. Gitlab also allows to run automatic and default pipelines (Auto DevOps) depending on the programming language used in the repository, but this functionality has been disabled to enforce the use of defined pipelines in the scope of the aerOS DevPrivSecOps methodology.

For self-hosted Gitlab installations, these runners need to be configured manually. First, a dedicated Linux VM was set up to install the Gitlab runner that will be responsible of running the different jobs that compose a Gitlab pipeline, which are always instantiated on demand according to the repository configuration. Before creating a runner, the associated executor must be selected. The Docker executor has been selected for the common runner in the aerOS project because it provides the full set of Gitlab executor capabilities such as a clean build environment for each build and easy runner migration to other machines in case of failure. Finally, a group runner was created in the Gitlab dashboard following the official documentation.

```
root@gitlab-runners:/home/eouser# gitlab-runner list
Runtime platform                                    arch=amd64 os=linux pid=1573818 revision=f5da3c5a version=16.6.1
Listing configured runners                          ConfigFile=/etc/gitlab-runner/config.toml
runner_1                                            Executor=docker Token=glrt-q_RAH8qDEFsAsn1gPqmg URL=https://gitlab.aeros-project.eu
```

Figure 6 aerOS common runner in the CF's dedicated VM

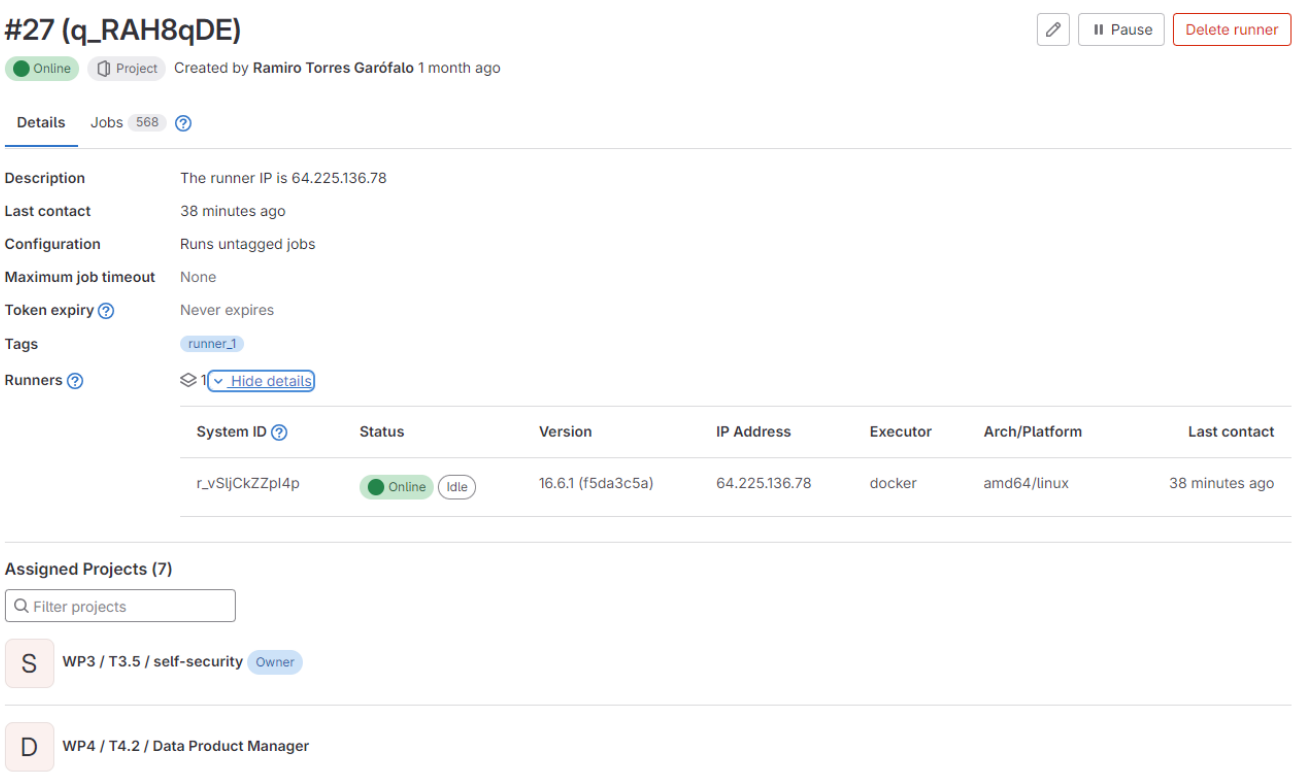

Figure 7 aerOS common runner configuration in the Gitlab dashboard.

3.3. SAST

Static Application Security Testing (SAST) refers to using automated tools for code analysis. The goal of SAST is not to replace manual code reviews but to be used in parallel to automate basic code checks. SAST helps in identifying vulnerabilities during the development phase in a short time and without the need for specialized personnel. SAST is designed to work in conjunction with other techniques like Dynamic Application Security Testing (DAST) and Software Composition Analysis (SCA) to ensure comprehensive application security throughout the development lifecycle. Integrating SAST into DevPrivSecOps can be particularly effective because it combines the strengths of both practices. SAST tools can be used as part of the automated CI/CD pipeline to scan new code commits for vulnerabilities. This integration ensures that every piece of code is tested for security before it is merged into the main branch and deployed. There are various opensource and commercial tools each one different from the others, but most of them perform two main tasks:

Transformation of the code into an abstract model: SAST tools generally take as input the source code of an application and convert it into an abstract representation for deeper analysis. Many tools in order to depict the code use Abstract Syntax Trees (AST), although some may use different, proprietary structures. This transformation is essential because it enables the analysis of code to be independent of language. This ensures that security vulnerabilities are not missed due to the nuances of language-specific features not being accurately captured.

Analysis of the abstract model for security issues: Different analysis techniques are used to search for potential vulnerabilities. The most relevant analysis techniques are:

1. Semantic analysis: This type of analysis is similar to using grep to search for potentially insecure functions during manual code reviews. Its goal is to identify vulnerabilities related to the use of potentially risky code.

2. Dataflow analysis: This approach focuses on examining how information moves from inputs to potentially risky functions. It tracks the path of data from where it enters the system to where it could cause harm.

3. Structural analysis: This method examines the specific code structures unique to each programming language. This process involves ensuring adherence to best practices in class declarations, identifying unreachable segments of code (known as dead code), properly utilizing try/catch blocks, and avoiding issues related to the use of insecure cryptographic elements like weak keys or initialization vectors

4. Configuration analysis: This process targets flaws in application settings rather than in the code itself. For instance, applications on Internet Information Services use a 'web.config' file, while PHP utilizes a 'php.ini' file for configuration options, with most applications employing some form of configuration file. By examining these configurations, the tool can pinpoint potential enhancements.

5. Control Fow analysis: This analysis evaluates the sequence of operations in the code to detect issues such as race conditions, the use of uninitialized variables, or resource leaks.

SAST can be implemented in various ways depending on the specific needs:

CI/CD Integration: SAST tools can scan for vulnerabilities each time a pull request or merge is executed. This ensures that code is preliminarily secured before it is merged. To avoid delays in the development pipeline, some projects may choose to run SAST scans only at the time of merges instead of on every pull request, preventing a backlog and waiting time for developers.

IDE Integration: Instead of waiting for a pull request or merge, SAST tools can also be integrated directly into the developers' preferred Integrated Development Environments (IDEs). This allows developers to identify and address issues early in the coding process, saving time and enhancing security throughout the project lifecycle.

GitLeaks is a SAST tool to identify secrets hardcoded in the programs that can lead to privacy and security problems. SonarQube is a widely used open-source tool for continuous inspection of code quality. It performs automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities. Semgrep, on the other hand, is a tool designed for performing lightweight static analysis that looks for patterns in code that may indicate security issues, bugs, or anti-patterns. The integration of SAST tools into a DevPrivSecOps framework within GitLab CI/CD pipelines represents a robust strategy for embedding security into the software development lifecycle. This approach not only enhances the security posture of applications but also aligns with modern practices of agile, continuous delivery, and automated deployments. As security threats evolve, so too should the strategies and tools used to combat them, making the continuous improvement aspect of DevPrivSecOps critical to the long-term resilience and success of software development projects.

3.3.1. Secret Scanning: GitLeaks

GitLeaks is a SAST tool for **detecting** and **preventing** hardcoded secrets like passwords, API keys, and tokens in git repositories. This tool allows us to analyse the privacy of the code by identifying any personal data (such as passwords) that is hardcoded in the code uploaded to the repository.

As well as having privacy implications, the inclusion of passwords, API keys and tokens in the code can open a security breach allowing potential attackers to gain access to the application by using them.

To deploy GitLeaks in the repository to be analysed, the configuration shown in Figure 8 needs to be added to the "gitlab-ci.yml" file.


```yaml
stages:
  - test

gitleaks:
  stage: test
  image:
    name: zricethezav/gitleaks
    entrypoint: [""]
  script:
    - gitleaks detect --verbose --source .
  rules:
    - if: '$CI_COMMIT_BRANCH == "testing-2"'
  allow_failure: true
```

Figure 8 GitLeaks configuration in GitLab.

As shown in the Figure 8, the configuration must define the GitLeaks docker image that must be used to implement the test, the command that must be launched (in the script section) and finally the rule that must be fulfilled to launch the test. In this case, the rule is that the repository to be monitored (testing-2) must have a commit. This rule will automatically launch the GitLeaks test every time a commit is made to the repository being monitored, thus analysing the hardcoded secrets.

3.3.2. SonarQube

SonarQube is a source code evaluation platform. It is a free software, and it uses several static source code analysis tools such as Checkstyle [15], PMD[16] or FindBugs [17] to obtain metrics that can help improve the quality of a code.

The continuous code quality inspection offered by SonarQube is used to perform automatic reviews with static code analysis to detect bugs and code smells in 19 programming languages. SonarQube provides reports on duplicate code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security recommendations.

For the aerOS project, it has been decided to deploy the Docker version of the community edition of SonarQube [18] on a server provisioned for these functions in the project.

SonarQube can be integrated with GitLab, via runners

Before the runner can be launched, it must be registered on the machine where SonarQube is installed, so that it can be linked to the GitLab repository where we want to launch the tests (Figure 9). The CI/CD section of the GitLab repository provides the commands and steps to run on the machine.

```
root@runner-sonarqube:~# gitlab-runner register
Runtime platform                                    arch=amd64 os=linux pid=2537 revision=3046fee8 version=16.6.0
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
https://gitlab.aeros-project.eu
Enter the registration token:
glrt-Y3fKe5e97KKS3Azrc4zt
Verifying runner... is valid                        runner=Y3fKe5e97
Enter a name for the runner. This is stored only in the local config.toml file:
[runner-sonarqube]: runner-sonarqube
Enter an executor: docker, parallels, ssh, docker+machine, instance, kubernetes, custom, docker-windows, shell, virtualbox, docker-autoscaler:
docker
Enter the default Docker image (for example, ruby:2.7):
alpine:latest
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
```

Figure 9 Installation of the runner in the SonarQube machine.

Finally, a project must be created using the SonarQube GUI that is associated with the GitLab repository to be analysed.

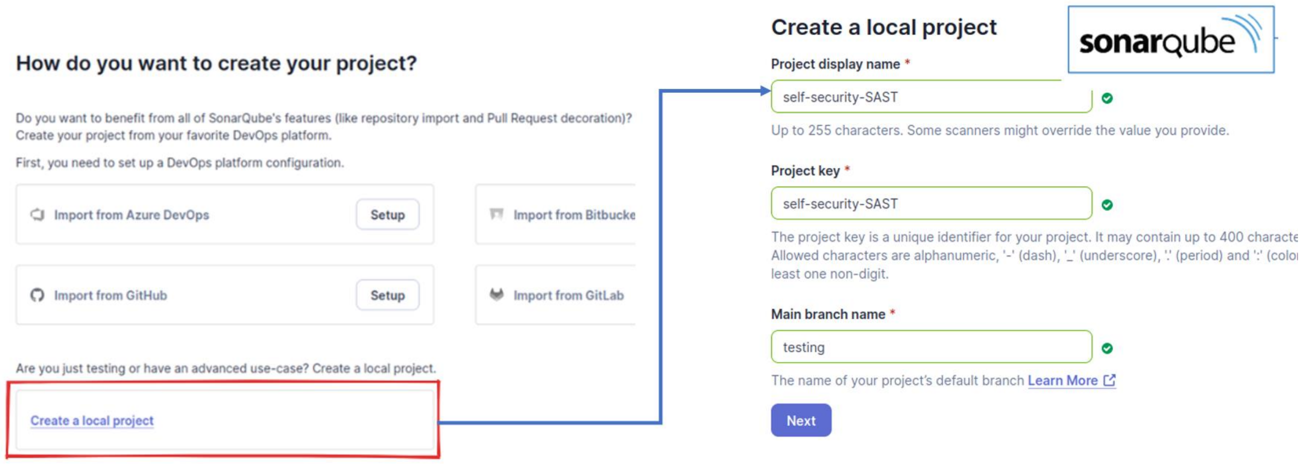

Figure 10 Creation of a project for a GitLab repository in SonarQube.

Once everything is installed, the environment variables must be added to the GitLab repository so that it can connect to SonarQube. Finally, a file with the SonarQube properties must be created in the repository.

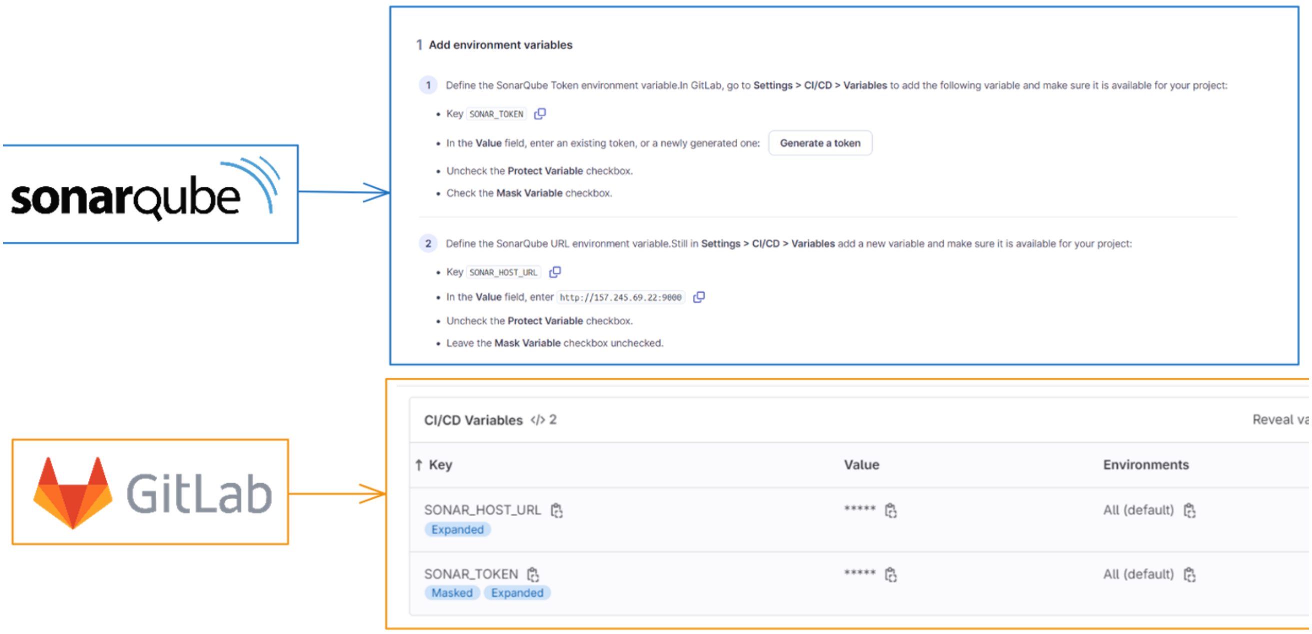

Figure 11 GitLab configuration for the connection with SonarQube.

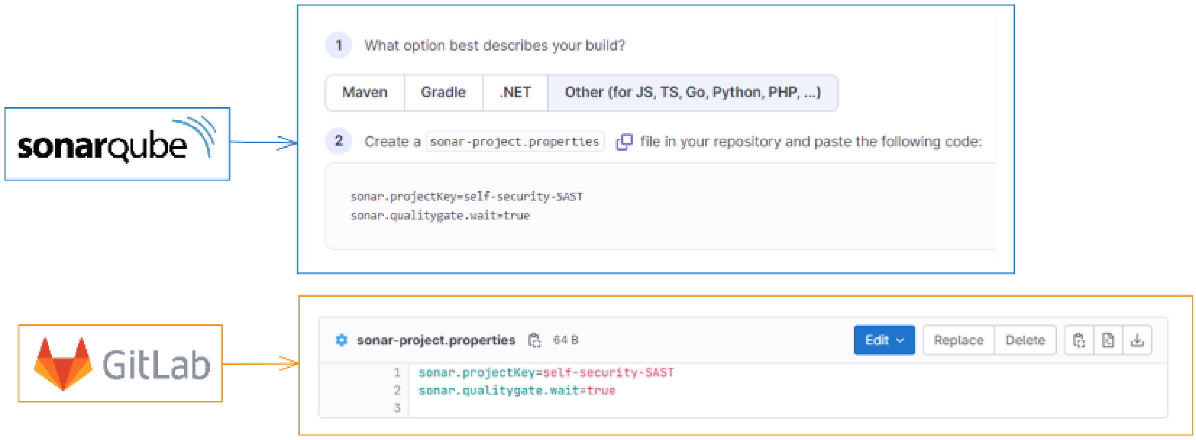

Figure 12 GitLab configuration for the connection with SonarQube(2).

Once the runner is configured, every time a commit is made to the repository selected to be analysed, the runner will automatically launch the SAST analysis by connecting to the SonarQube server. The result of the analysis is displayed in GitLab and is also shown in SonarQube's SonarQube user interface (Figure 13).

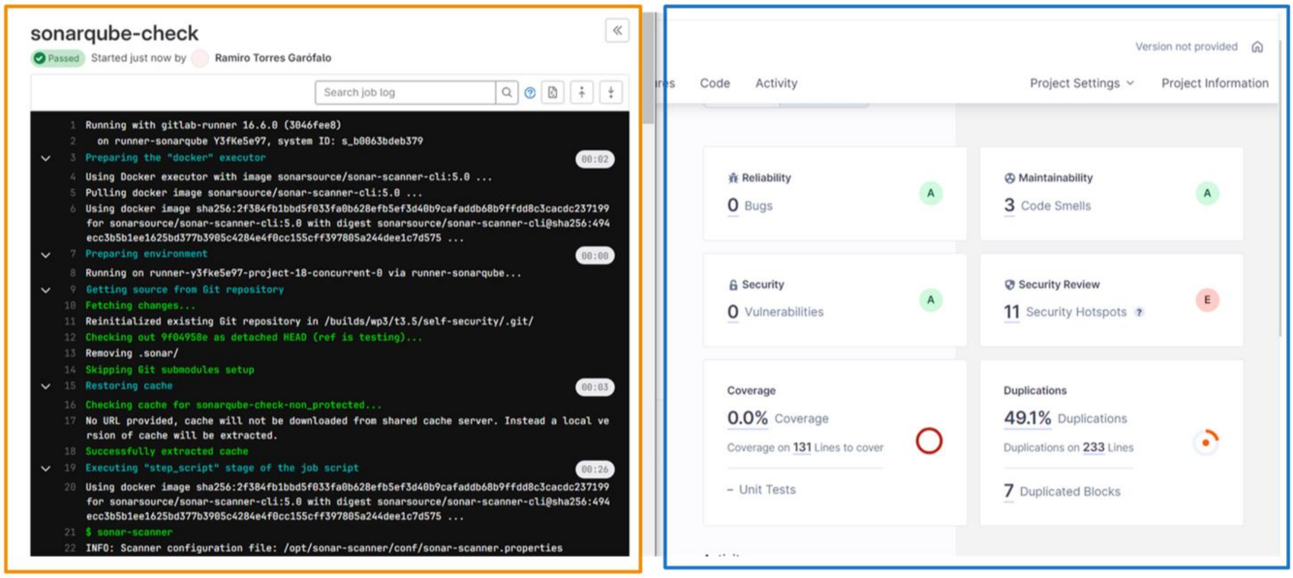

Figure 13 Successful analysis result in GitLab and also in SonarQube.

3.3.3. Semgrep

Semgrep is an open-source static analysis tool used to help improve the quality of code by detecting bugs, enforcing coding standards and identifying security vulnerabilities. It’s a semantic code grepper, meaning it can search code for patterns that look like bugs or coding violations, rather than just matching exact strings. It works with a variety of programming languages, including Python, JavaScript, Go, Java, Ruby and Typescript, and it aims to support more languages in the future.

Semgrep performs scans on a whole project either on-demand or automatically during every build or commit in CI/CD, with all analysis conducted locally. In aerOS, to comply with the principles of DevPrivSecOps, it will be integrated with Gitlab using runners to scan the project’s source code for issues.

As in the SonarQube deployment, we must link the runner to the repository that we want to use. The next step is the registration process that allows the runner to access and execute the Gitlab CI/CD pipeline defined in the repository. This pipeline includes the steps to run tests using Semgrep. The Figure 14 hows the runner registration on the system hosting Semgrep to run the build jobs and send the results back to Gitlab.

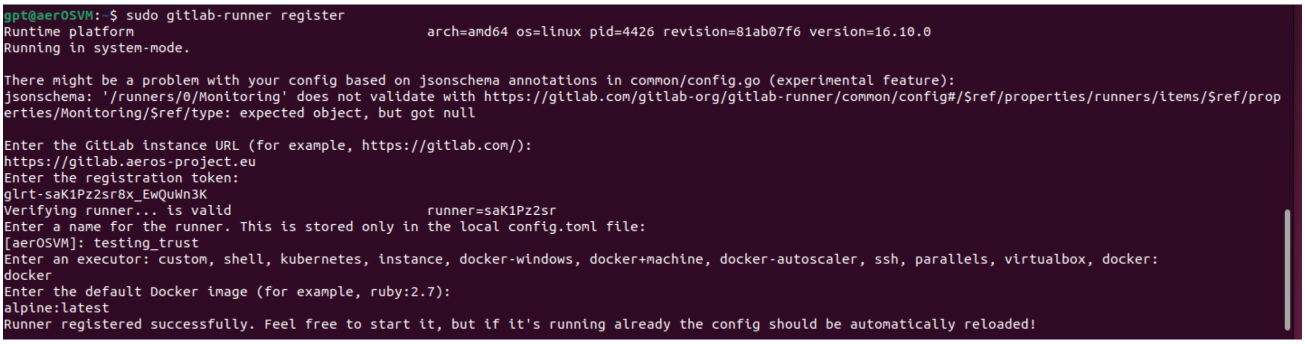

Figure 14 Registration of the runner in the Semgrep machine.

Once this is done, we can now move on to configuring the GitLab CI/CD pipeline. In this phase, we need to edit the '.gitlab-ci.yml' file to run SAST using Semgrep. This file allows developers to write specific instructions for GitLab CI/CD on how to build and test their application. If GitLab CI/CD is enabled, every commit to the repository automatically triggers the pipeline described in this file. This configuration specifies what should happen in the GitLab CI/CD pipeline when the code is pushed or merged.

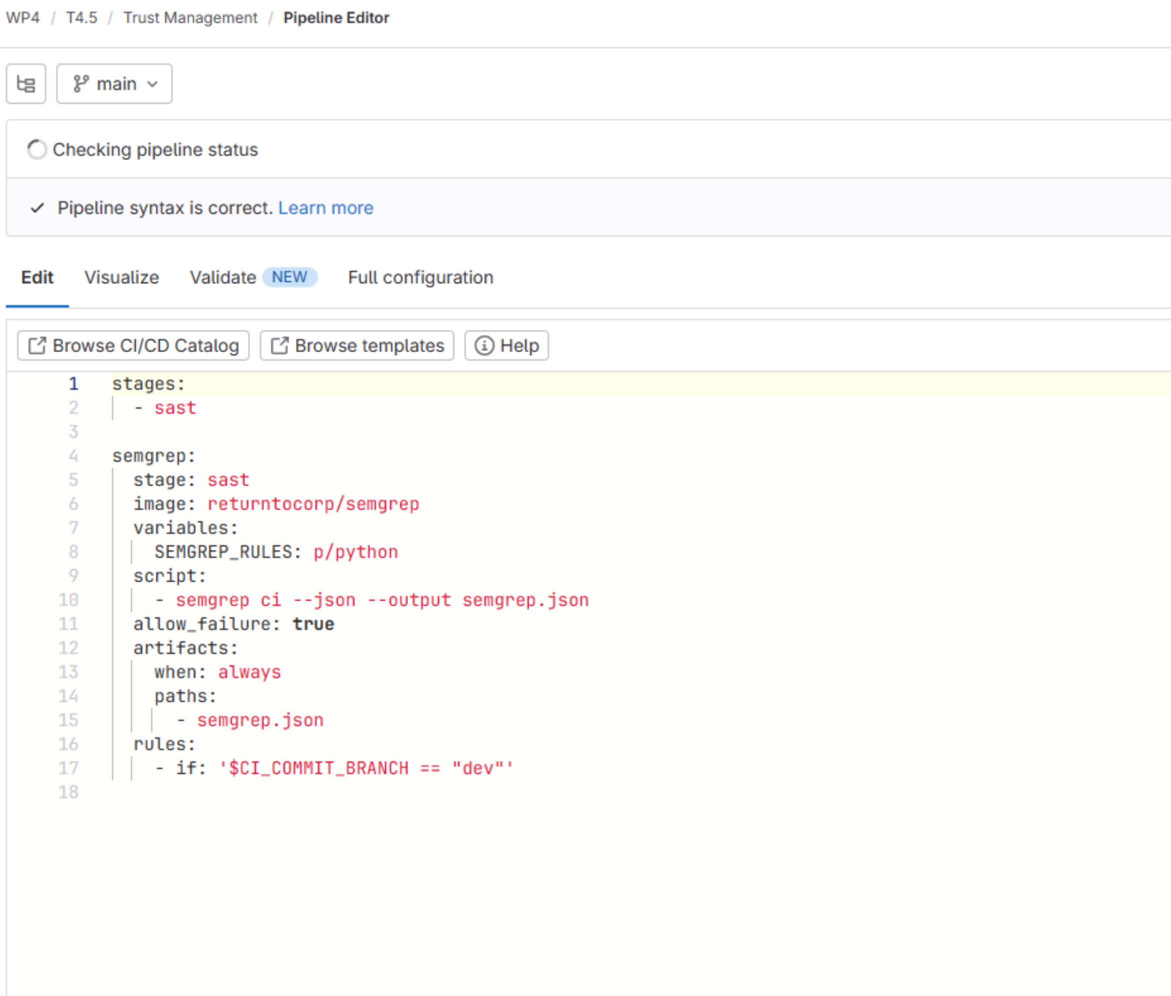

Figure 15 Gitlab CI/CD configuration using Semgrep.

In the Figure depicted above, a job named 'semgrep' has been established within a SAST stage, utilizing the Semgrep tool in a pre-built Docker container to scan Python code for security issues. The results of the scan are stored in the semgrep.json file. This job is specifically configured to run only when changes are made to the ‘dev’ branch. The pipeline will not fail even if Semgrep finds issues, ensuring continuous integration flow remains uninterrupted.

Once the changes to the '.gitlab-ci.yml' file are committed, the runner will automatically initiate the SAST analysis using the returntocorp/semgrep Docker image to execute Semgrep scans. Figure 16 demonstrates how Semgrep is executed in our repository.

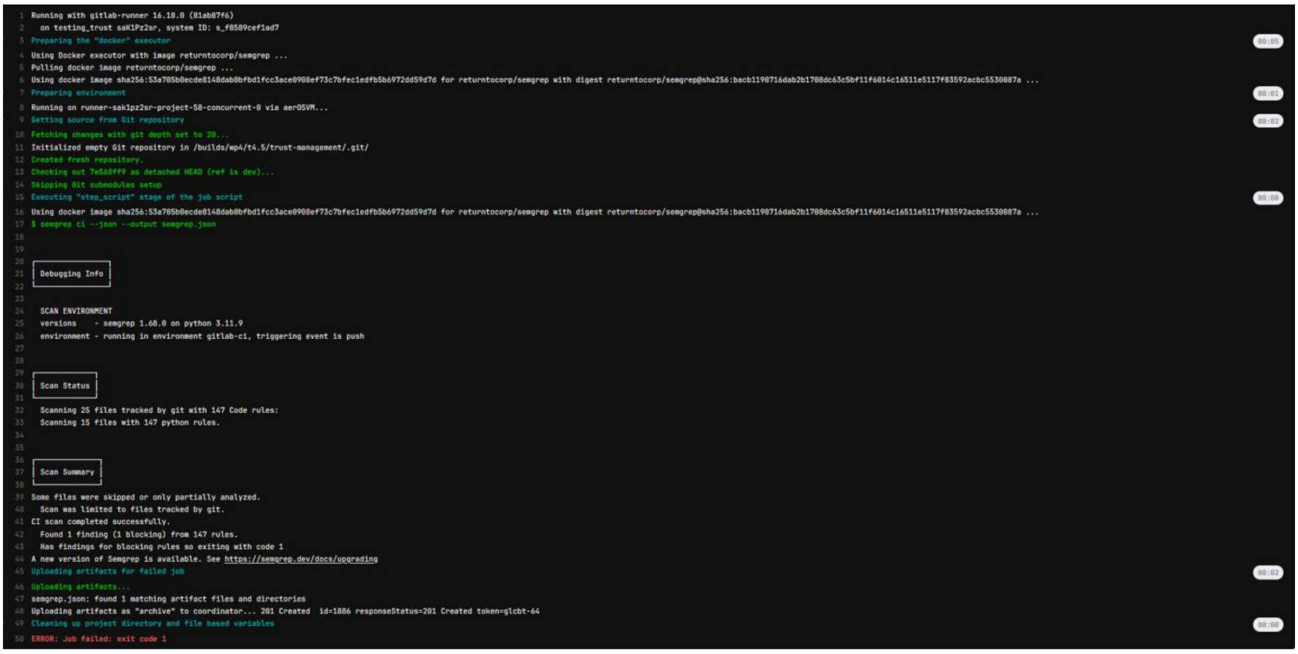

Figure 16 The process of execution Semgrep in our repository.

3.4. Container Scanning: Trivy

Although containers provide an efficient and scalable way to develop and deploy applications, they can also present a risk if not operated correctly. Container images that have not been verified or analysed may contain malware, unpatched vulnerabilities or insecure configurations that can be exploited by malicious actors.

For this reason, container image analysis should be an essential element in the software development lifecycle. By systematically scanning and analysing container images prior to deployment, we can identify and remediate any potential threats, thus ensuring a secure and robust environment.

Trivy is an open-source container vulnerability scanning tool. It is very effective at finding vulnerabilities in both container images and dependencies in source code projects. These are the strengths of Trivy:

- **Container image scanning:** Trivy can scan container images for vulnerabilities. It supports several container image formats, including Docker and Kubernetes. It can detect vulnerabilities in operating system packages and libraries included in container images.
- **Project dependency analysis**: Trivy can also scan-source code projects for dependencies. It supports many programming languages and package managers.
- **Vulnerability database**: Trivy maintains an open-source vulnerability database that is regularly updated. This means that we can obtain information about the latest known vulnerabilities.

In order to include the Trivy analysis in the GitLab CI/CD pipeline, it is necessary to add the following lines (Figure 17) in the ".gitlab-ci.yml" file, once the image of the container is builded.

```
scan_image_self-security:
  stage: scan_image
  needs: ["build_image_self-security"]
  image: docker:24
  services:
    - name: docker:24-dind
      alias: docker
  before_script:
    - apk --no-cache add curl python3 py3-pip
    - curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin $CI_REGISTRY
  script:
    - docker pull "${REGISTRY_IMAGE_PATH_SURICATA}:latest"
    - trivy image --exit-code 1 "${REGISTRY_IMAGE_PATH_SURICATA}:latest"
  rules:
    - if: '$CI_COMMIT_BRANCH == "testing-2"'
```

Figure 17 Trivy inclusion in GitLab CI/CD.

Once the container image has been generated, Trivy, which has been deployed on the machine where the GitLab runner is configured, will scan this newly created image. It will scan for any vulnerabilities that the image may have, and it will also scan the dependencies that the image has, trying to find any known vulnerabilities in its updated database.

3.5. Deployment automation: Flux CD

FluxCD is a set of Continuous Delivery (CD) tools and solutions for Kubernetes. CD is a pattern focused on producing, releasing and deploying software in short cycles, following pipelines to ensure it is built, tested and released faster and more frequently.

This allows to reduce risk and for a more constant delivery of updates, thus testing the changes in a more controlled and incremental way.

FluxCD provides a GitOps Toolkit to follow the GitOps approach, a way of implementing CD. The concept of GitOps is to have a Git repository that always contains declarative descriptions of the desired state of the infrastructure/application in the production environment and an automated process to ensure that the production environment matches such state. GitOps allows the developer to deploy or update an application by simply updating the associated repository, and the automated process handles the rest, saving time and securing that the repository adequately describes the current state of the application, acting as a single source of truth.

For the aerOS project, the FluxCD solution has been chosen to handle the CD process, due to it fitting with the current Kubernetes infrastructure of the project.

There is a set of core concepts to learn about to properly understand how FluxCD operates, extensively explained in [20].

A **Source** defines the origin of a repository containing the desired state of the system and the requirements to obtain it. For example, a certain branch, a tag, etc. Sources are checked for changes on a defined interval and, if there is a newer version available, an **artefact** is produced.

Artifacts are objects that describe the changes and are consumed by other Flux component to perform actions, such as updating the cluster, to match the desired state.

Reconciliation refers to the process of ensuring that an application within the cluster or infrastructure matches a state as defined in the Source.

A **Kustomization** represents a set of Kubernetes resources that Flux is supposed to **reconcile** in the cluster.

A **Bootstrap** is the process of installing the Flux components in a GitOps manner.

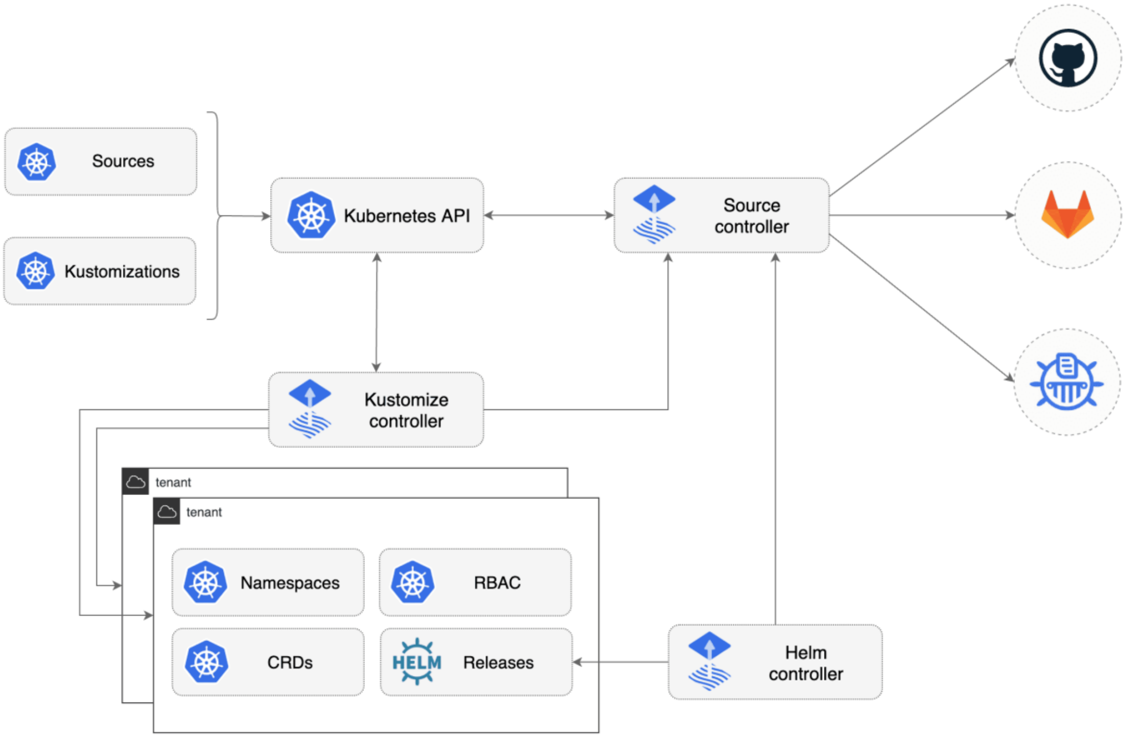

Figure 18 FluxCD GitOps Toolkit.

FluxCD can be integrated with GitLab by applying the following steps[19].

The process begins by generating a GitLab Access Token **with read and write permissions to the API** by accessing the User settings/Access tokens and selecting to create a new token, then export it in the system.

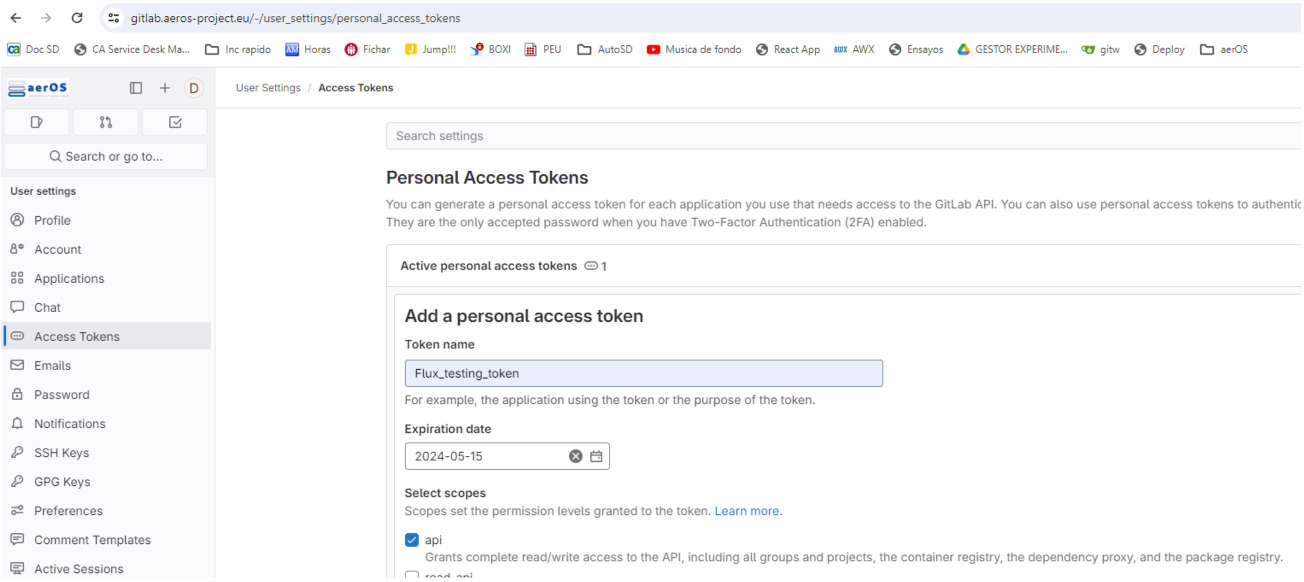

Figure 19 Create Access Token.

```
export GITLAB_TOKEN =<your-token>
```

Before proceeding, it can be checked that everything is ready to be ran and deployed by flux with `flux check`.

```
root@Ubuntu-desktop:/home/ubuntu# flux check
► checking prerequisites
✔ Kubernetes 1.29.2 >=1.26.0-0
► checking version in cluster
✔ distribution: flux-v2.2.3
✔ bootstrapped: false
► checking controllers
✔ helm-controller: deployment ready
► ghcr.io/fluxcd/helm-controller:v0.37.4
✔ kustomize-controller: deployment ready
► ghcr.io/fluxcd/kustomize-controller:v1.2.2
✔ notification-controller: deployment ready
► ghcr.io/fluxcd/notification-controller:v1.2.4
✔ source-controller: deployment ready
► ghcr.io/fluxcd/source-controller:v1.2.4
► checking crds
✔ alerts.notification.toolkit.fluxcd.io/v1beta3
✔ buckets.source.toolkit.fluxcd.io/v1beta2
✔ gitrepositories.source.toolkit.fluxcd.io/v1
✔ helmcharts.source.toolkit.fluxcd.io/v1beta2
✔ helmreleases.helm.toolkit.fluxcd.io/v2beta2
✔ helmrepositories.source.toolkit.fluxcd.io/v1beta2
✔ kustomizations.kustomize.toolkit.fluxcd.io/v1
✔ ocirepositories.source.toolkit.fluxcd.io/v1beta2
✔ providers.notification.toolkit.fluxcd.io/v1beta3
✔ receivers.notification.toolkit.fluxcd.io/v1
✔ all checks passed
```

Figure 20 Flux check.

With the token ready, the next step is to run the GitLab bootstrap with the command `flux bootstrap gitlab.`

```
flux bootstrap gitlab \
  --token-auth \
  --hostname=my-gitlab-enterprise.com \
  --owner=my-gitlab-group \
  --repository=my-project \
  --branch=master \
  --path=clusters/my-cluster
```

As an example, here is the configuration used for the aerOS project flux test.

```
flux bootstrap gitlab \
  --token-auth \
  --hostname=gitlab.aeros-project.eu \
  --owner=p26-nasertic \
  --repository=flux-test \
  --branch=master \
  --path=./clusters/my-cluster
```

If everything worked correctly, the Controller pods should have deployed within the Kubernetes cluster. It may be verified by checking the pods in the `flux-system` namespace.

```
root@Ubuntu-desktop:/home/ubuntu# kubectl get pods -n flux-system
NAME                                       READY   STATUS    RESTARTS      AGE
helm-controller-5d8d5fc6fd-g8d6b           1/1     Running   2 (26m ago)   53m
kustomize-controller-7b7b47f459-cmlrp      1/1     Running   0             34m
notification-controller-5bb6647999-6kdlw   1/1     Running   0             53m
source-controller-7667765cd7-qm88h         1/1     Running   2 (26m ago)   53m
```

Figure 21 Deployed controller pods.

The Flux repository should have also been created with the manifests pushed to GitLab.

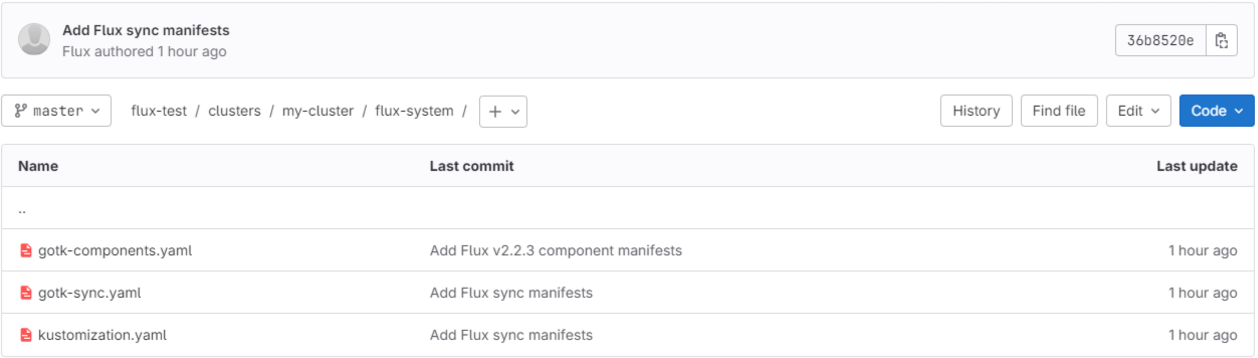

Figure 22 Flux repository.

3.6. DAST: ZAP

Dynamic Application Security Testing (DAST) involves examining a live web application for vulnerabilities by simulating an attacker's approach. It plays a crucial role in the DevPrivSecOps lifecycle. This black-box testing method aims to identify and exploit weaknesses as an actual attacker would, using either manual efforts or automated tools. DAST tests the application from an external perspective, allowing testers to disregard the internal mechanisms of the application and focus on pinpointing vulnerabilities that are most likely to be exploited by attackers. The insights from DAST typically highlight critical vulnerabilities that do not require insider knowledge to exploit, thereby prioritizing issues that need immediate attention. Although DAST is a valuable tool in the security testing, it does not replace other security testing methods. instead, it enhances them.

Integrating DAST into the development pipeline is essential for ensuring application security throughout various phases of software development, and in real-time. CI/CD pipelines streamline the software delivery process by automating tasks like code compilation, running tests, and deploying to production environments. Due to its need for a running application, DAST is optimally used within CI/CD pipelines, where it can effectively test applications in environments that mimic live production settings.

In the DevPrivSecOps, DAST tools are typically employed at multiple stages, especially during and after the development phase. These tools are designed for quick feedback, unlike traditional web application vulnerability scanners that might take hours to complete. This real-time analysis is crucial for identifying security weaknesses that might not be apparent during static analysis or code review.

OWASP ZAP (Zed Attack Proxy) is an open-source web application security scanner. Developed by the Open Web Application Security Project (OWASP), it is one of the most popular tools used for testing web applications, helping to identify security vulnerabilities during the development and testing phases of software development cycles.

ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually. It is designed to be user-friendly for beginners in application security, yet powerful enough for professional penetration testers. The most important key features of ZAP are:

- **Automated Scanner**: ZAP can perform both active and passive scanning. Active scanning involves ZAP automatically testing the application for vulnerabilities using known attack patterns. Passive scanning, on the other hand, monitors website traffic and analyses it for signs of potential security issues.
- **Intercept Proxy**: This function turns ZAP into a man-in-the-middle-proxy between the tester's browser and the web application, allowing the tester to intercept, inspect, and modify the traffic passing through.
- **Spider**: It uses a spider to crawl websites and automatically discover new resources (URLs).
- **Scanning**: It can passively scan traffic that passes through it without altering it, and it can also perform active scanning which sends modified data to the server to check for vulnerabilities.

- **REST API**: ZAP includes a REST API for interacting with the tool programmatically, which is useful for integrating ZAP into Continuous Integration namely CI pipelines.

OWASP ZAP is planned to be integrated into aerOS' GitLab repository to scan a running application in accordance with DevPrivSecOps regulations. Like the deployment of the SAST tools, it is necessary to link the runner with the repository intended for use. The registration of the runner will be foregone, as the runner previously registered during the Semgrep deployment will be utilized for testing purposes.

Proceeding with the configuration of the GitLab CI/CD pipeline. In this phase, it is necessary to modify the '.gitlab-ci.yml' file to implement DAST using OWASP ZAP. This file enables developers to specify detailed instructions for GitLab CI/CD on how to build and test their application.

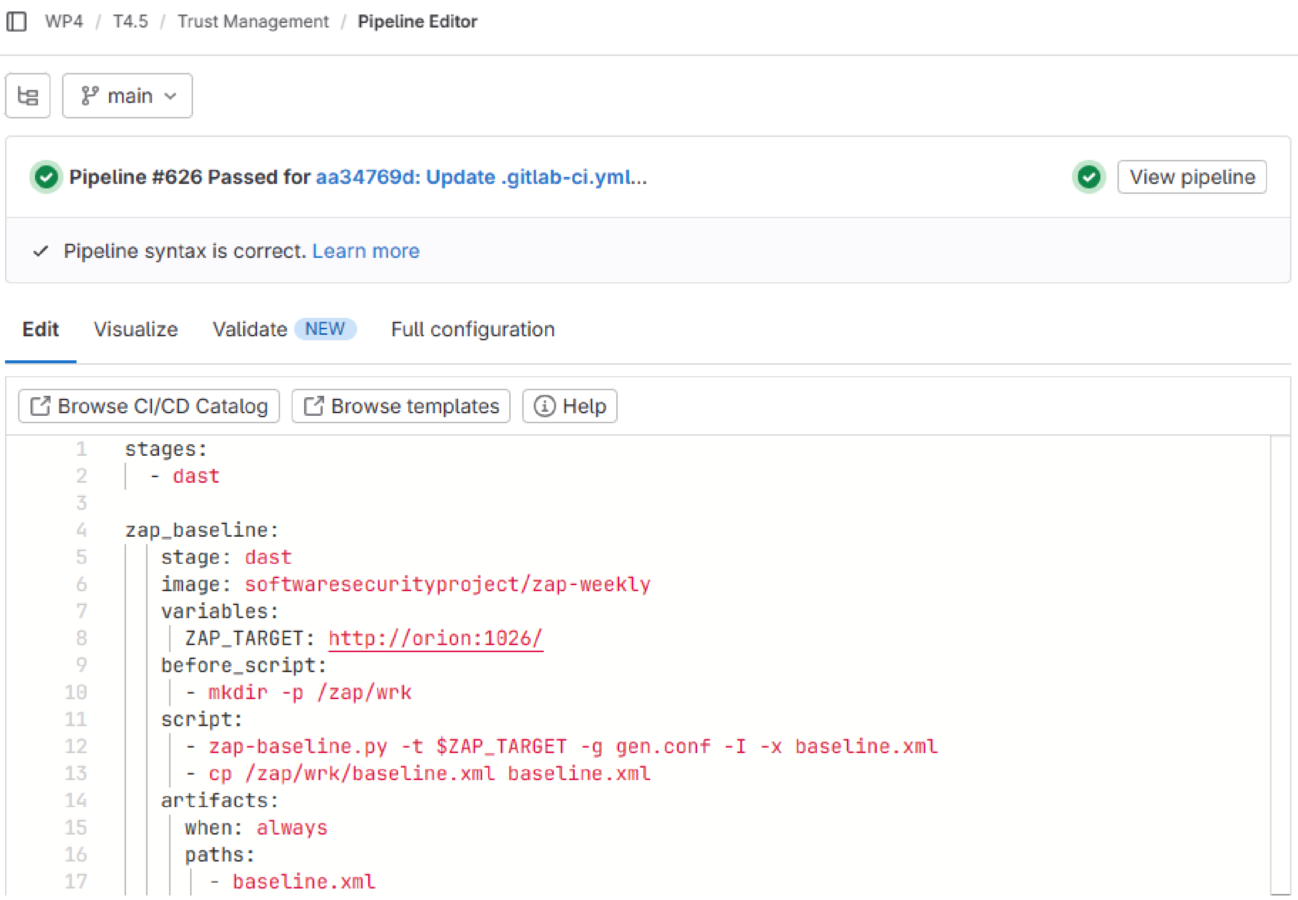

Figure 23 Gitlab CI/CD configuration using ZAP.

Analysing what is depicted in the above figure, within the "dast" stage, there is a job named "zap_baseline" configured to run using the "softwaresecurityproject/zap-weekly" Docker image.

The configuration sets an environment variable, ZAP_TARGET, which specifies the target URL of the web application to be tested. In this instance, the target is the ORION-LD Context Broker, which is running locally on our premises at the URL http://orion:1026/.

The artifacts section specifies that the baseline.xml file should always be saved as an artifact of the build, regardless of whether the job succeeds or fails. This file is essential for reviewing the security findings of the scan and for potentially integrating those results into further stages of the development and security assessment processes.

After committing the changes to the '.gitlab-ci.yml' file, the runner will automatically begin the DAST analysis using the OWASP ZAP Docker image to perform automated scans. Figure 24 illustrates the execution of ZAP in our repository.

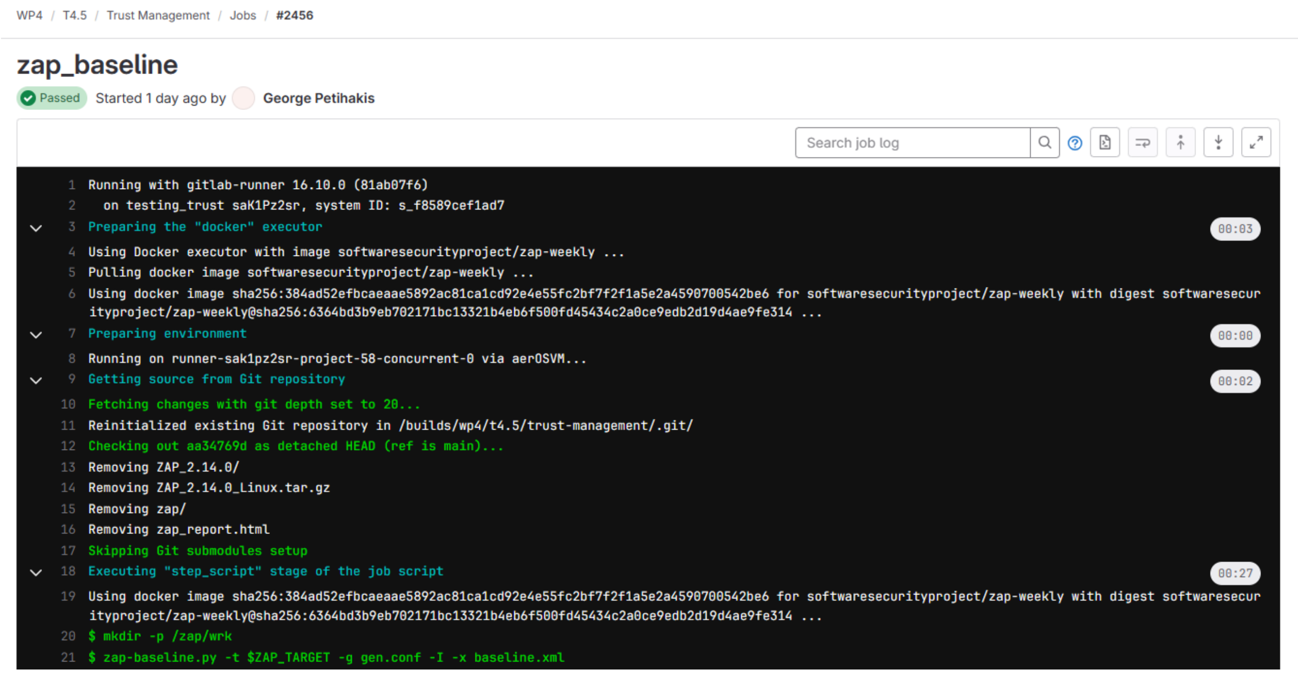

Figure 24 Tthe execution process of ZAP in the aerOS repository.

After executing OWASP ZAP, no vulnerabilities were detected in our running application, which attests to the robust structure of our code. In the following figure, the output from 'baseline.xml' is displayed, which contains the results of the security scan, viewed through an online reader

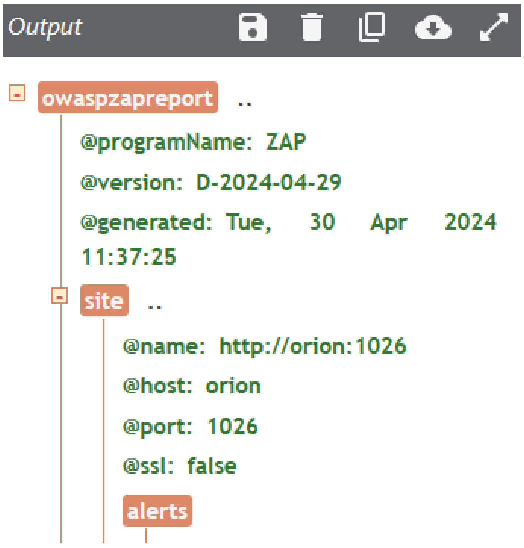

Figure 25 The output of the baseline.xml through an online reader.

The report header contains metadata about the report itself, this report was generated on Tuesday, 30 April 2024 at 11:37:25.

The body of the XML shows details about the specific site that was scanned, identified by the URL "http://orion:1026". The 'site' tag includes attributes such as 'host' and 'port', which are set to "orion" and "1026" respectively, and 'ssl' set to "false", indicating that the site does not use SSL encryption.

The <alerts> tag within the 'site' section is empty, signifying that no security vulnerabilities were found during the scan of this particular site.

In general, OWASP ZAP effectively scans web applications to detect potential security threats. The lack of alerts in the report implies that the configurations and aspects of the tested web application are secure under the scanning conditions.

3.7. Privacy: GDPR compliance Checklist

The need to analyse privacy during the software development lifecycle has led aerOS to develop a GDPR compliance tool in the software development lifecycle. This tool is based on the GDPRchecklist [21] tool and aims to analyse whether the tools developed in the project and specifically the tools used in the pilotss comply with this rule. This tool has been modified to meet the specific needs of the aerOS project.

Privacy and especially GDPR compliance is directly related to data and data usage. Therefore, aerOS aims, through this checklist to guide the developers and users of the pilots to comply with this regulation.

The tool is intended to guide developers in the correct use of private data in the different components that interact with them.

For end users, in the case of the pilots in the project, the aim is to analyse the types of data to be used and how they should be used correctly.

This tool has also recommendations on how to comply with the different sections of the GDPR law, thus guiding developers and users to comply with it.

As can be seen in image Figure 26, the use of data within the project is classified into three groups, depending on the different roles that are in charge of the data management:

- *Data controller*: the module in charge of controlling the data processing. In the case of the aerOS project, the Data Fabric oversees managing the continuum data.
- *Data Processor*: the ML models or data processing tools of the pilots are part of this group in the aerOS project.
- *Data subject*: in this case, the data provider is analysed. In the context of the project, all data sources that are connected to the aerOS continuum are analysed in this section.

These roles are assigned to each of the questions asked in the different groups of the checklist: data, accountability and management, new rights, special cases and user rights.

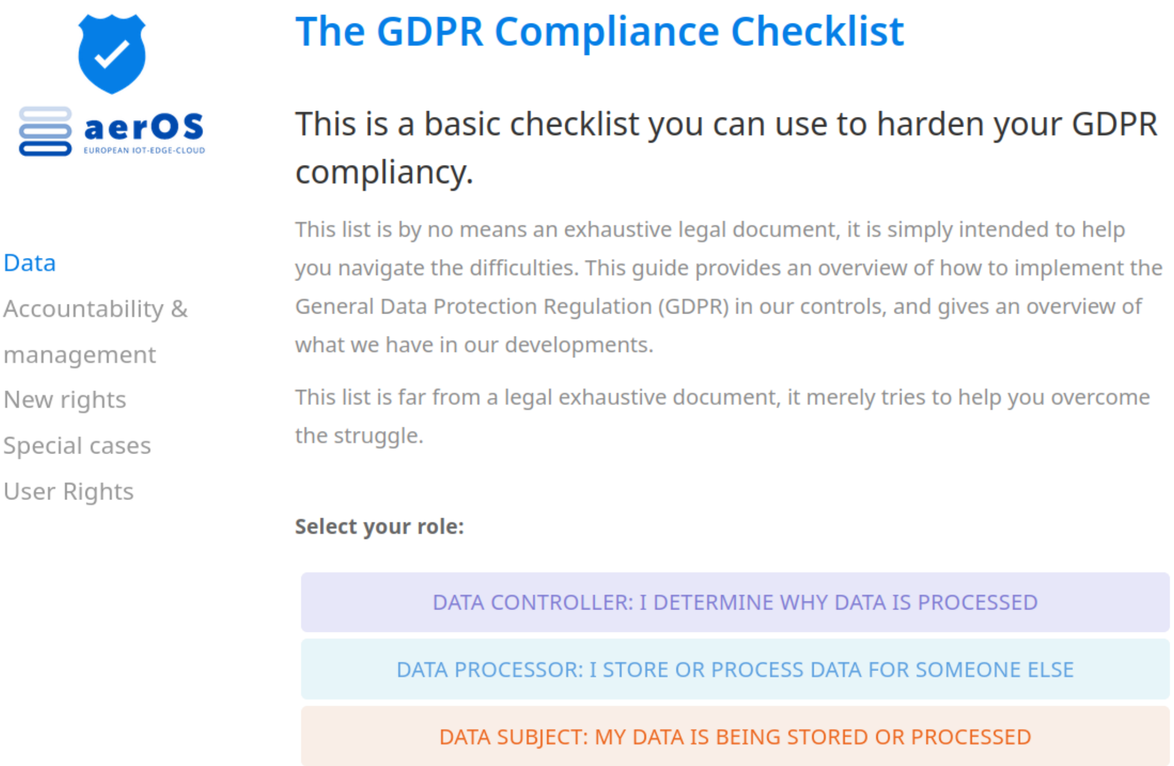

Figure 26 aerOS GDPR compliance checklist.

The checklist is divided into 4 main sections: Data, Accountability and management, new rights and user rights.

- **Data**: By means of this analysis, the aim is to see whether the processing of the pilot data by the Data Fabric complies with the GDPR regulation, or if something needs to be changed in this tool to make it compliant (Figure 27).
- **Accountability and management:** In this section it is analysed that the environment, in this case aerOS, is safe and that any problems are reported to the authorities correctly. This is measured with the checkboxes shown in Figure 28.
- **New rights:** By means of the checklist shown in Figure 29, it is intended to ensure that aerOS users, and in particular data providers to the continuum, are always in control of their data and can modify or delete it at any time.
- **User rights:** This section analyses that the rights of the providers of the data used in the continuum are always complied with. Figure 30 shows the sections to be fulfilled by aerOS to comply with the GPDR regulation.

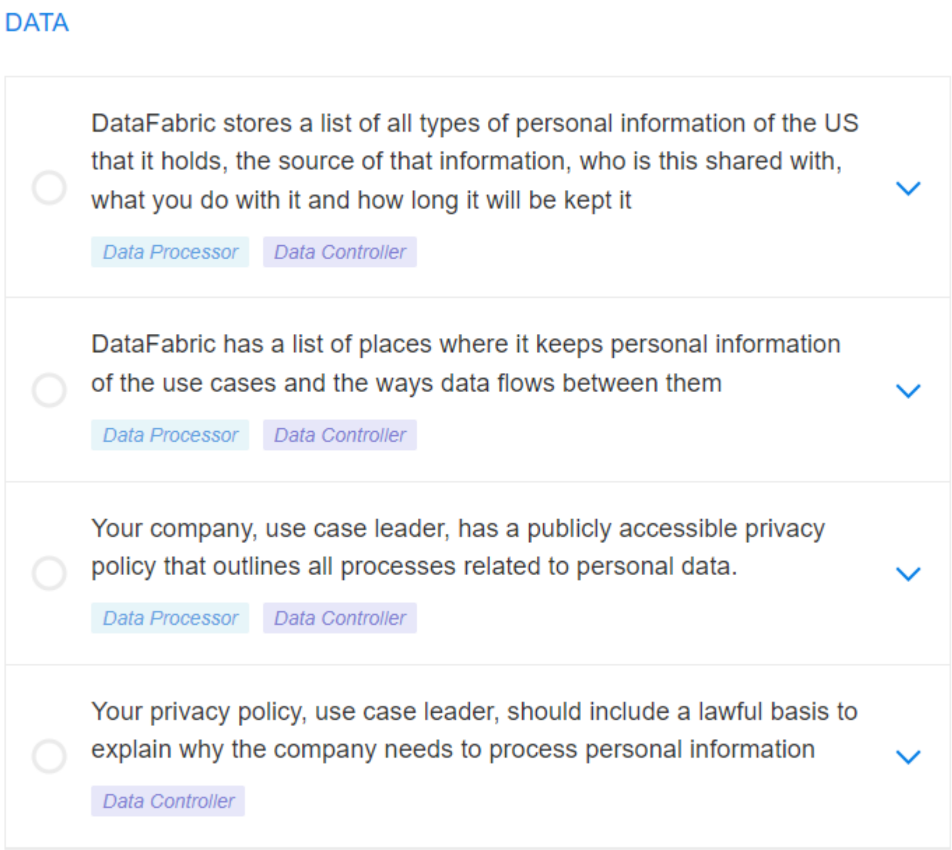

Figure 27 Analysis of the GDPR in the data.

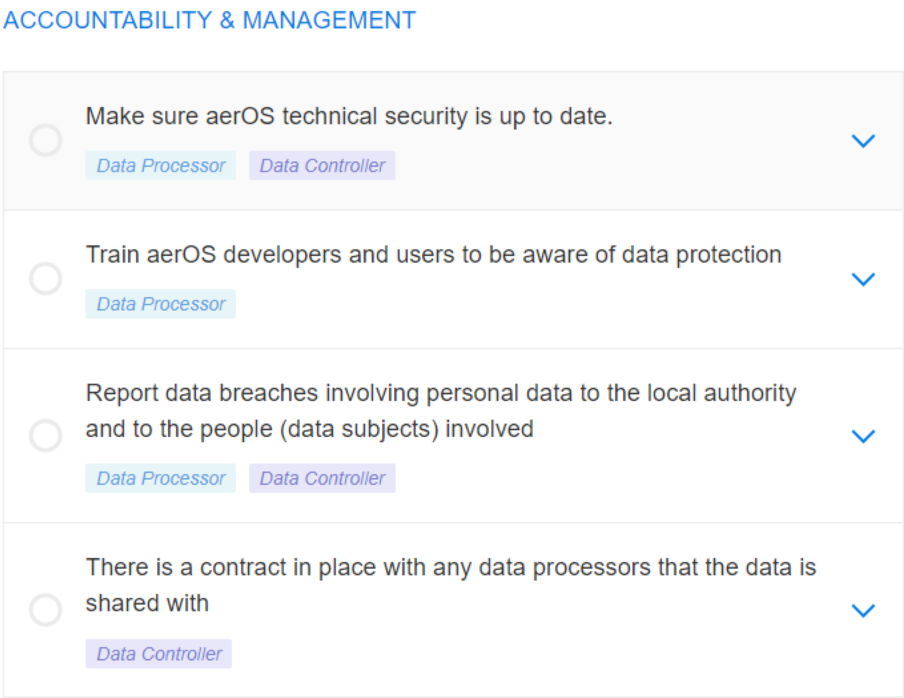

Figure 28 Analysis of the GDPR in Accountability and management.

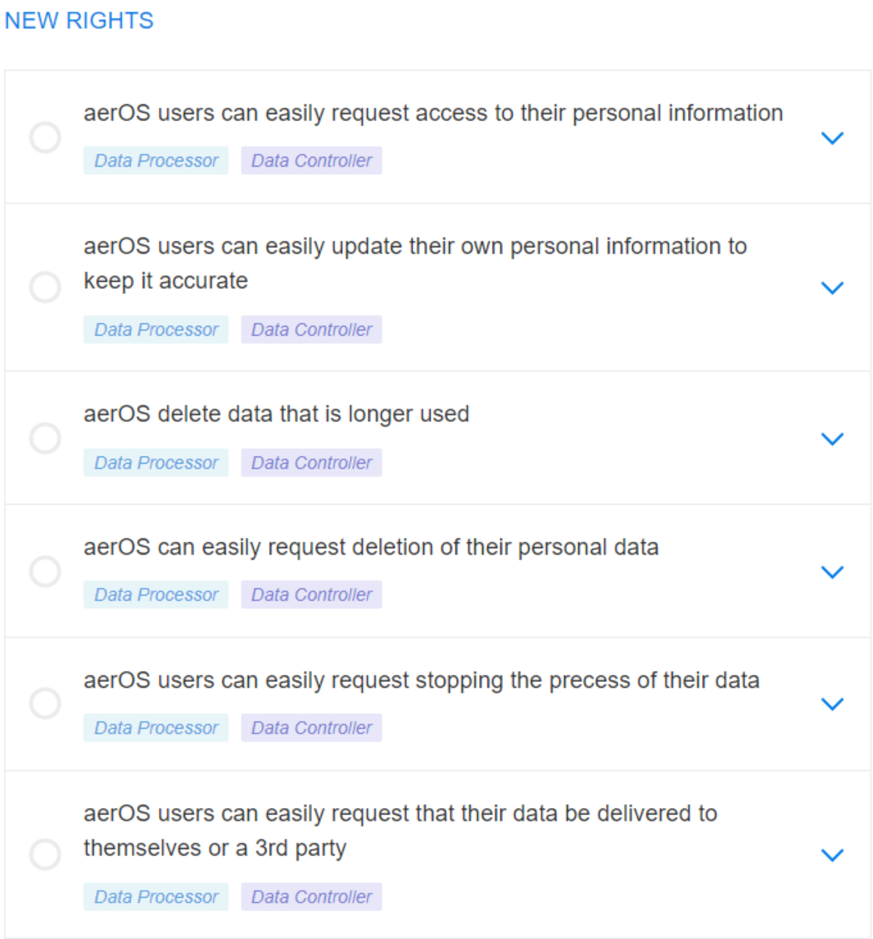

Figure 29 Analysis of the GDPR in new rights.

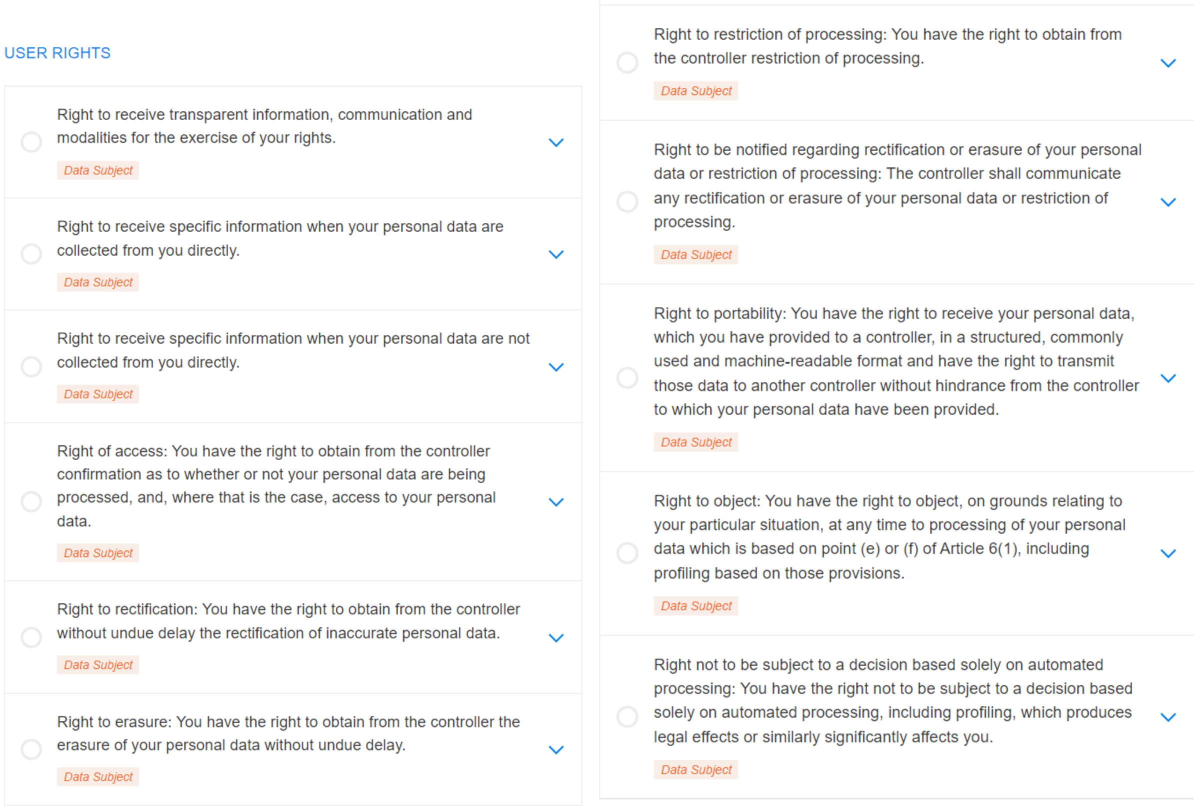

Figure 30 Analysis of the GDPR in user rights.

This tool will be distributed to all developers together with the 3 cookbooks in the month M21. It will also be distributed to the leaders of each pilot so that they can start analysing the use of the data in the project and take the necessary actions before starting to implement aerOS in the pilots. In addition, in WP5 the pilots will be tracked using this tool until the end of the project to ensure that they are all GDPR compliant.

4. DevPivSecOps implementation example

The tools presented in the previous step have been put in place to complete a pipeline. This pipeline has been created with interconnections between the different tools, creating steps and dependencies between them. These dependencies allow to block the execution of the pipeline if one of the tests has not been passed.

By testing a code with the pipeline, and if it has completed all the steps successfully, it allows us to ensure that the code that has been deployed is secure and privacy aware.

Figure 31shows an example of the pipeline where the different steps that have been designed to carry out the aerOS DevPrivSecOps methodology can be seen.

Figure 31 aerOS CI/CD pipeline in GitLab.

GitLab is the tool that orchestrates the execution of the different steps of the pipeline. The result obtained in the execution of the different tools in each of these steps can also be visualised in GitLab.

If one of the steps of the pipeline has not been completed satisfactorily, the report that the tools leave in GitLab must be analysed and the necessary modifications made to the code until the entire pipeline is executed satisfactorily. The error log is saved as an artefact by GitLab. This provides information about the cause of the problem detected.

An example of a CI/CD pipeline where the aerOS DevPrivSecOps methodology is implemented is presented in Annex A. It is intended to implement a pipeline for each module developed in the project, following this example presented in the annex.

This execution together with the GDPR compliance analysis through the checklist will allow to ensure that all the code generated and deployed in aerOS complies with the defined security and privacy requirements.

5. Conclusion

The methodology presented in this document, together with the guidelines of the usage of each of the selected tools, will allow aerOS and other developers to build secure and privacy aware by design software components, thus meeting the needs of the market.

Emphasis should be placed on the fact that through the different controls that have been implemented in the methodology for analysing the privacy compliance of the developed code, the state of the art has been improved.

References

[1] “Secure by design” [Online]. Available: https://www.cisa.gov/securebydesign#:~:text=Secure%20by%20Design%20principles%20should,for%20broad%20use%20or%20consumption. [Accessed 28th May 2024]

[2] “Zero-day attacks in 2023” [Online]. Available: https://blog.google/technology/safety-security/a-review-of-zero-day-in-the-wild-exploits-in-2023/ [Accessed 28th May 2024]

[3] “GDPR-info” [Online]. Available: https://gdpr-info.eu [Accessed 28th May 2024]

[4] Schwartz, P. M. (2019). Global data privacy: The EU way. NYUL Rev., 94, 771.

[5] Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *Ieee Software*, *33*(3), 94-100.

[6] “What are the Phases of DevSecOps?” [Online]. Available: https://www.veritis.com/blog/what-are-the-phases-of-devsecops/ [Accessed 28th May 2024]

[7] “GitLab” [Online]. Available: https://about.gitlab.com/ [Accessed 28th May 2024]

[8] “GitLeaks” [Online]. Available: https://github.com/gitleaks/gitleaks [Accessed 28th May 2024]

[9] “Semgrep” [Online]. Available: https://semgrep.dev/ [Accessed 28th May 2024]

[10] “SonarQube” [Online]. Available: https://www.sonarsource.com/products/sonarqube/ [Accessed 28th May 2024]

[11] “Retire.js” [Online]. Available: https://github.com/RetireJS/retire.js [Accessed 28th May 2024]

[12] “Trivy” [Online]. Available: https://trivy.dev/ [Accessed 28th May 2024]

[13] “Flux CD” [Online]. Available: https://fluxcd.io/ [Accessed 28th May 2024]

[14] “ZAP” [Online]. Available: https://www.zaproxy.org/ [Accessed 28th May 2024]

[15] “Checkstyle” [Online]. Available: https://checkstyle.sourceforge.io/ [Accessed 28th May 2024]

[16] “PMD” [Online]. Available: https://pmd.github.io/ [Accessed 28th May 2024]

[17] “FindBugs” [Online]. Available: https://findbugs.sourceforge.net/ [Accessed 28th May 2024]

[18] “SonarQube docker version” [Online]. Available: https://hub.docker.com/_/sonarqube [Accessed 28th May 2024]

[19] “GitLab Flux CD integration”. Available: https://fluxcd.io/flux/installation/bootstrap/gitlab/ [Accessed 28th May 2024]

[20] “Flux documentation” [Online]. Available: https://fluxcd.io/flux/cmd/flux_create_secret_git/ [Accessed 28th May 2024]

[21] “GDPR Checklist” [Online]. Available: https://gdprchecklist.io/ [Accessed 28th May 2024]

A. aerOS DevPrivSecOps CI/CD configuration example

In this appendix we have included an example of a CI/CD pipeline configuration that allows the implementation of the DevPrivSecOps methodology defined in task T2.4. It is expected that developers will use this pipeline as an example to implement in the project's code repositories.

```
variables:
  GIT_SUBMODULE_STRATEGY: recursive
  PUSH_IMAGE: "true"
  BUILD_ARGS: "kafka sqlite"
  SCAN_IMAGE: "false"
  EXTRA_TAGS: "${CI_COMMIT_TAG}"
  CONTEXT_DIR_ETL: "k8s-infra/Docker_image_files/etl"
  REGISTRY_IMAGE_PATH_ETL: "${CI_REGISTRY}/wp3/t3.5/self-security/etl"
  CONTEXT_DIR_SURICATA: "k8s-infra/Docker_image_files/suricata"
  REGISTRY_IMAGE_PATH_SURICATA: "${CI_REGISTRY}/wp3/t3.5/self-security/self-security"
  DOCKER_FILE_NAME: "Dockerfile"
  DOCKER_HOST: "tcp://docker:2375"
  DOCKER_TLS_CERTDIR: ""

stages:
  - secret_scanning
  - sast
  - sca
  - build_image
  - scan_image
  - deploy
  - dast

gitleaks:
  stage: secret_scanning
  image:
    name: zricethezav/gitleaks
    entrypoint: [""]
  script:
    - gitleaks detect --verbose --source . -f json -r detect_gitleaks.json
  allow_failure: true
  artifacts:
    when: always
    paths:
      - detect_gitleaks.json
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

gitleaks_protect:
```

```
  stage: secret_scanning
  image:
    name: zricethezav/gitleaks
    entrypoint: [""]
  script:
    - gitleaks protect --verbose --source . -f json -r protect_gitleaks.json
  allow_failure: true
  artifacts:
    when: always
    paths:
      - protect_gitleaks.json
  dependencies:
    - gitleaks
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

njsscan:
  stage: sast
  needs: ["gitleaks_protect"]
  image: python
  before_script:
    - pip3 install --upgrade njsscan
  script:
    - njsscan --exit-warning . --sarif -o njsscan.sarif
  allow_failure: true
  artifacts:
    when: always
    paths:
      - njsscan.sarif
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

semgrep:
  stage: sast
  needs: ["gitleaks_protect"]
  image: returntocorp/semgrep
  variables:
    SEMGREP_RULES: p/javascript
  script:
    - semgrep ci --json --output semgrep.json
  allow_failure: true
  artifacts:
    when: always
    paths:
      - semgrep.json
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'
```

```
sonarqube-check:
  stage: sast
  needs: ["gitleaks_protect"]
  image:
    name: sonarsource/sonar-scanner-cli:5.0
    entrypoint: [""]
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
    GIT_DEPTH: "0"
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
    - sonar-scanner
  allow_failure: true
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

sonarqube-vulnerability-report:
  stage: sast
  script:
    - apt-get update && apt-get install -y curl
                        -            'curl           -u           "${SONAR_TOKEN}:"
"${SONAR_HOST_URL}/api/issues/gitlab_sast_export?projectKey=selft-
security&branch=${CI_COMMIT_BRANCH}&pullRequest=${CI_MERGE_REQUEST_IID}"   -o   gl-
sast-sonar-report.json'
  allow_failure: true
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'
  artifacts:
    expire_in: 1 day
    reports:
      sast: gl-sast-sonar-report.json
  dependencies:
    - sonarqube-check

retire:
  stage: sca
  needs: ["gitleaks_protect"]
  script:
    - echo "SCA ..."

build_image_self-security:
  stage: build_image
  needs: ["njsscan","semgrep","sonarqube-vulnerability-report","retire"]
  image:
    name: gcr.io/kaniko-project/executor:v1.14.0-debug
```

```
    entrypoint: [""]
  script:
    - |
      if [[ -z "${REGISTRY_USER}" || -z "${REGISTRY_PASSWORD}" ]]; then
        REGISTRY_USER=${CI_REGISTRY_USER}
        REGISTRY_PASSWORD=${CI_REGISTRY_PASSWORD}
        unset CI_REGISTRY_USER; unset CI_REGISTRY_PASSWORD;
      fi
    - |
      if [ -z "${REGISTRY_IMAGE_PATH_SURICATA}" ]; then
        echo "ERROR: CI variable REGISTRY_IMAGE_PATH_SURICATA is mandatory."
        exit 1
      fi
    - REGISTRY=$(echo ${REGISTRY_IMAGE_PATH_SURICATA} | cut -d / -f 1)
    - |
      KANIKO_CONTEXT_DIR_SURICATA=${CI_PROJECT_DIR}/${CONTEXT_DIR_SURICATA}
    - mkdir -p /kaniko/.docker
    - |
              echo    "{\"auths\":{\"${REGISTRY}\":{\"auth\":\"$(printf    "%s:%s"
"${REGISTRY_USER}"     "${REGISTRY_PASSWORD}"     |     base64     -w0)\"}}}"     >
/kaniko/.docker/config.json
    - |
      if [ "$(echo ${PUSH_IMAGE} | tr '[:upper:]' '[:lower:]')" = "true" ]; then
        PUSH_IMAGE=""
      else
        echo "Info: defer pushing image to remote as PUSH_IMAGE is false"
        PUSH_IMAGE="--no-push"
      fi
    - |
      if [ -n "$EXTRA_TAGS" ]; then
        IMAGE_WITHOUT_TAG=$(echo ${REGISTRY_IMAGE_PATH_SURICATA} | cut -d : -f 1)
        for tag in $EXTRA_TAGS; do
                          KANIKO_EXTRA_TAGS="${KANIKO_EXTRA_TAGS}    --destination
${IMAGE_WITHOUT_TAG}:${tag}"
        done
      fi
    - /kaniko/executor
      --context "${KANIKO_CONTEXT_DIR_SURICATA}"
      --dockerfile "${KANIKO_CONTEXT_DIR_SURICATA}/${DOCKER_FILE_NAME}"
      --build-arg optional_dependencies="${BUILD_ARGS}"
           --destination   "${REGISTRY_IMAGE_PATH_SURICATA}"   ${KANIKO_EXTRA_TAGS}
${PUSH_IMAGE}
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

build_image_etl:
  stage: build_image
  needs: ["njsscan","semgrep","sonarqube-vulnerability-report","retire"]
```

```
  image:
    name: gcr.io/kaniko-project/executor:v1.14.0-debug
    entrypoint: [""]
  script:
    - |
      if [[ -z "${REGISTRY_USER}" || -z "${REGISTRY_PASSWORD}" ]]; then
        REGISTRY_USER=${CI_REGISTRY_USER}
        REGISTRY_PASSWORD=${CI_REGISTRY_PASSWORD}
        unset CI_REGISTRY_USER; unset CI_REGISTRY_PASSWORD;
      fi
    - |
      if [ -z "${REGISTRY_IMAGE_PATH_ETL}" ]; then
        echo "ERROR: CI variable REGISTRY_IMAGE_PATH_ETL is mandatory."
        exit 1
      fi
    - REGISTRY=$(echo ${REGISTRY_IMAGE_PATH_ETL} | cut -d / -f 1)
    - |
      KANIKO_CONTEXT_DIR_ETL=${CI_PROJECT_DIR}/${CONTEXT_DIR_ETL}
    - mkdir -p /kaniko/.docker
    - |
              echo    "{\"auths\":{\"${REGISTRY}\":{\"auth\":\"$(printf    "%s:%s"
"${REGISTRY_USER}"     "${REGISTRY_PASSWORD}"     |     base64     -w0)\"}}}"     >
/kaniko/.docker/config.json
    - |
      if [ "$(echo ${PUSH_IMAGE} | tr '[:upper:]' '[:lower:]')" = "true" ]; then
        PUSH_IMAGE=""
      else
        echo "Info: defer pushing image to remote as PUSH_IMAGE is false"
        PUSH_IMAGE="--no-push"
      fi
    - |
      if [ -n "$EXTRA_TAGS" ]; then
        IMAGE_WITHOUT_TAG=$(echo ${REGISTRY_IMAGE_PATH_ETL} | cut -d : -f 1)
        for tag in $EXTRA_TAGS; do
                          KANIKO_EXTRA_TAGS="${KANIKO_EXTRA_TAGS}    --destination
${IMAGE_WITHOUT_TAG}:${tag}"
        done
      fi
    - /kaniko/executor
      --context "${KANIKO_CONTEXT_DIR_ETL}"
      --dockerfile "${KANIKO_CONTEXT_DIR_ETL}/${DOCKER_FILE_NAME}"
      --build-arg optional_dependencies="${BUILD_ARGS}"
      --destination "${REGISTRY_IMAGE_PATH_ETL}" ${KANIKO_EXTRA_TAGS} ${PUSH_IMAGE}
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

scan_image_self-security:
  stage: scan_image
```

```
  needs: ["build_image_self-security"]
  image: docker:24
  services:
    - name: docker:24-dind
      alias: docker
  before_script:
    - apk --no-cache add curl python3 py3-pip
    - curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin $CI_REGISTRY
  script:
    - docker pull "${REGISTRY_IMAGE_PATH_SURICATA}:latest"
    - trivy image --exit-code 1 "${REGISTRY_IMAGE_PATH_SURICATA}:latest"
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

scan_image_etl:
  stage: scan_image
  needs: ["build_image_etl"]
  image: docker:24
  services:
    - name: docker:24-dind
      alias: docker
  before_script:
    - apk --no-cache add curl python3 py3-pip
    - curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin $CI_REGISTRY
  script:
    - docker pull "${REGISTRY_IMAGE_PATH_ETL}:latest"
    - trivy image --exit-code 1 "${REGISTRY_IMAGE_PATH_ETL}:latest"
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

deploy_self-security:
  stage: deploy
  needs: ["build_image_self-security", "scan_image_self-security"]
  script:
    - echo "Deploy self-Security ..."
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

deploy_etl:
  stage: deploy
```

```
  needs: ["build_image_etl", "scan_image_etl"]
  script:
    - echo "Deploy ETL ..."
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

zap_baseline_etl:
  stage: dast
  needs: ["deploy_etl"]
  image: ghcr.io/zaproxy/zaproxy:stable
  script:
    - echo "zap"
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

zap_full_etl:
  stage: dast
  needs: ["deploy_etl"]
  image: ghcr.io/zaproxy/zaproxy:stable
  script:
    - echo "zap"
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

zap_baseline_self-security:
  stage: dast
  needs: ["deploy_self-security"]
  image: ghcr.io/zaproxy/zaproxy:stable
  variables:
    ZAP_TARGET: "http://URL:PORT/"
  before_script:
    - mkdir -p /zap/wrk
  script:
    - zap-baseline.py -t $ZAP_TARGET -g gen.conf -I -x baseline.xml
    - cp /zap/wrk/baseline.xml baseline.xml
  artifacts:
    when: always
    paths:
      - baseline.xml
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

zap_full_self-security:
  stage: dast
  needs: ["deploy_self-security"]
  image: ghcr.io/zaproxy/zaproxy:stable
  variables:
    ZAP_TARGET: "http://URL:PORT/"
```

```
before_script:
  - mkdir -p /zap/wrk
script:
  - zap-full-scan.py -t $ZAP_TARGET -g gen.conf -I -x full-zap.xml
  - cp /zap/wrk/full-zap.xml full-zap.xml
artifacts:
  when: always
  paths:
    - full-zap.xml
rules:
  - if: '$CI_COMMIT_BRANCH == "develop"'
```