esprit
Se former autrement

Rapport DevSecOps

Security in CI/CD

Realiser Par : Ben Bouali Dhia

2025-2026

30 novembre 2025

Table des matières

Chapitre 1

Introduction

1.1 Contexte du projet

Exemple etudiée : application en Node Js
Ce projet a été choisi comme base expérimentale pour déployer une démarche DevSecOps, car il représente une architecture Java moderne et modulaire, tout en étant volontairement conçu avec des vulnérabilités afin de servir de plateforme de test et d'analyse de sécurité.

1.2 Objectif du projet

Mise en place d'une démarche DevSecOps
L'objectif du projet est d'intégrer la sécurité dès les premières phases du cycle de développement selon le principe du Security Shift-Left. Il consiste à transformer une chaîne CI/CD traditionnelle en un pipeline DevSecOps, dans lequel les analyses et contrôles de sécurité sont automatisés à chaque étape du processus, depuis l'écriture du code jusqu'au déploiement.

1.3 Architecture globale de la solution

L'ensemble de cette architecture est déployé dans un environnement virtuel basé sur Ubuntu. Elle repose sur les composants suivants :

- **Code source** : hébergé sur GitHub.
- **Visual Studio Code** : environnement de développement.
- **Plugin SonarQube for VS Code** : analyse statique intégrée à l'IDE.
- **Serveur Jenkins** : orchestration du pipeline CI/CD.
- **SonarQube** : outil d'analyse statique du code (SAST).
- **Trivy** : scanner d'images Docker et analyse de composants (SCA).
- **Nikto** : outil d'analyse dynamique du code (DAST)
- **Webhook GitHub → Jenkins** : déclenchement automatique des builds et analyses.
- **Webhook Slack** : notifications automatiques des builds, scans et résultats d'analyses.

1.4 Scope du projet

Ce projet couvre la mise en place d'un pipeline DevSecOps complet, incluant :

- l'intégration continue (CI/CD) des applications,
- l'automatisation des contrôles et analyses de sécurité,
- la surveillance et la notification des builds et résultats de scans,
- l'analyse et la gestion des vulnérabilités détectées dans le code et l'infrastructure.

Chapitre 2

Realisation

2.1 Introduction du pipeline DevSecOps

Le pipeline DevSecOps mis en place automatise l'ensemble du cycle de développement, depuis l'écriture du code jusqu'au déploiement, en intégrant la sécurité à chaque étape (*Security Shift-Left*). Il combine les outils de gestion de code, d'intégration continue, d'analyse statique et dynamique, de vérification des dépendances, ainsi que des notifications automatisées.

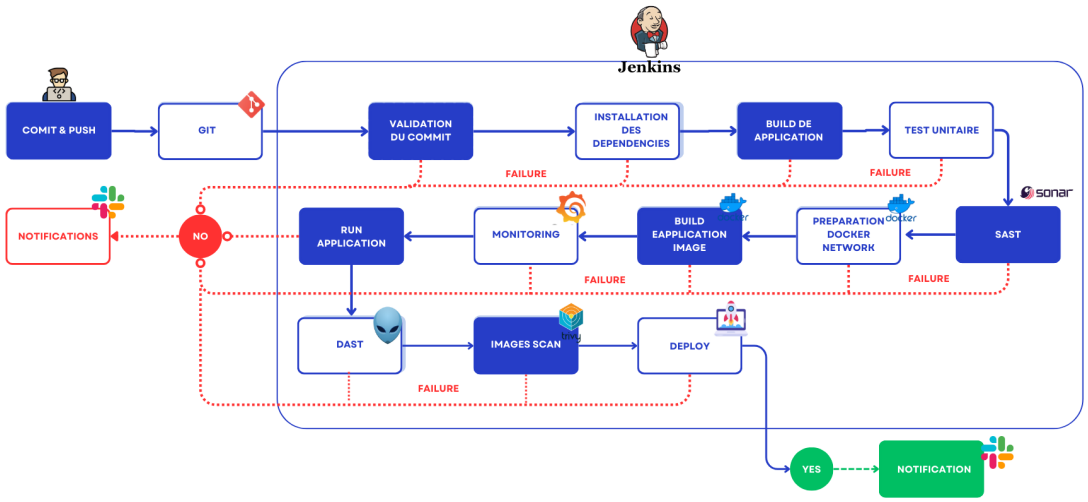

FIGURE 2.1 – Pipeline DevSecOps

La figure suivante illustre l'exécution réussie du pipeline DevSecOp.

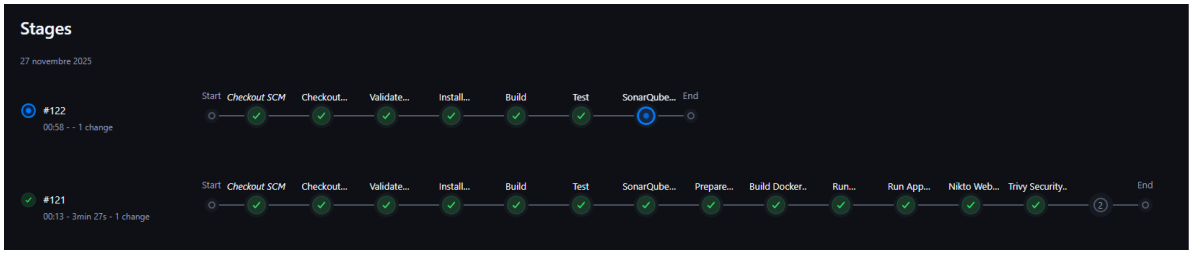

FIGURE 2.2 – Pipeline Jenkins

2.2 Stages

2.2.1 Récupération du code source (Checkout Code)

Dans cette étape, le pipeline Jenkins récupère le code source depuis le dépôt GitHub du projet. La branche spécifiée est main et l'accès au dépôt est sécurisé à l'aide des identifiants configurés dans Jenkins (credentialsId). Cette étape garantit que le pipeline travaille toujours sur la version la plus récente du code source avant de lancer les étapes de construction, de test et de déploiement.

FIGURE 2.3 – Checkout code

2.2.2 Validation du message de commit

Cette étape vérifie que le message du dernier commit respecte la convention définie pour le projet, ici la présence du mot-clé devsecops lab. Si le message ne correspond pas, le pipeline est interrompu pour éviter de traiter du code non conforme, et une notification est envoyée sur Slack pour informer l'équipe. Si le message est correct, le pipeline continue son exécution et une notification de validation réussie est envoyée.

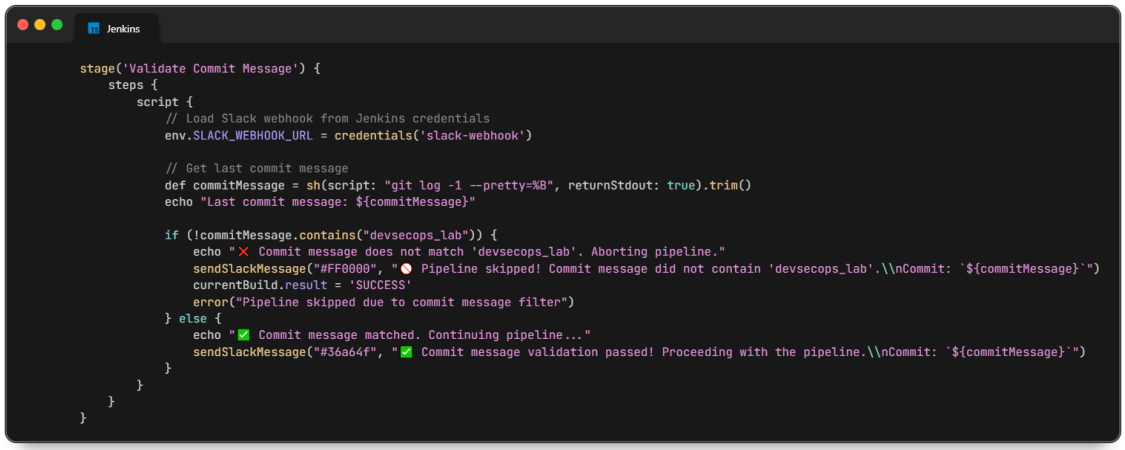

FIGURE 2.4 – Validation du message de commit

La fonction suivante permet d'envoyer des notifications personnalisées sur Slack depuis le pipeline Jenkins. Elle prend en entrée une couleur (color) et un texte (text) pour formater le message. Le texte est sécurisé en échappant les caractères spéciaux, puis un fichier JSON est généré et envoyé via la webhook Slack configurée dans Jenkins.

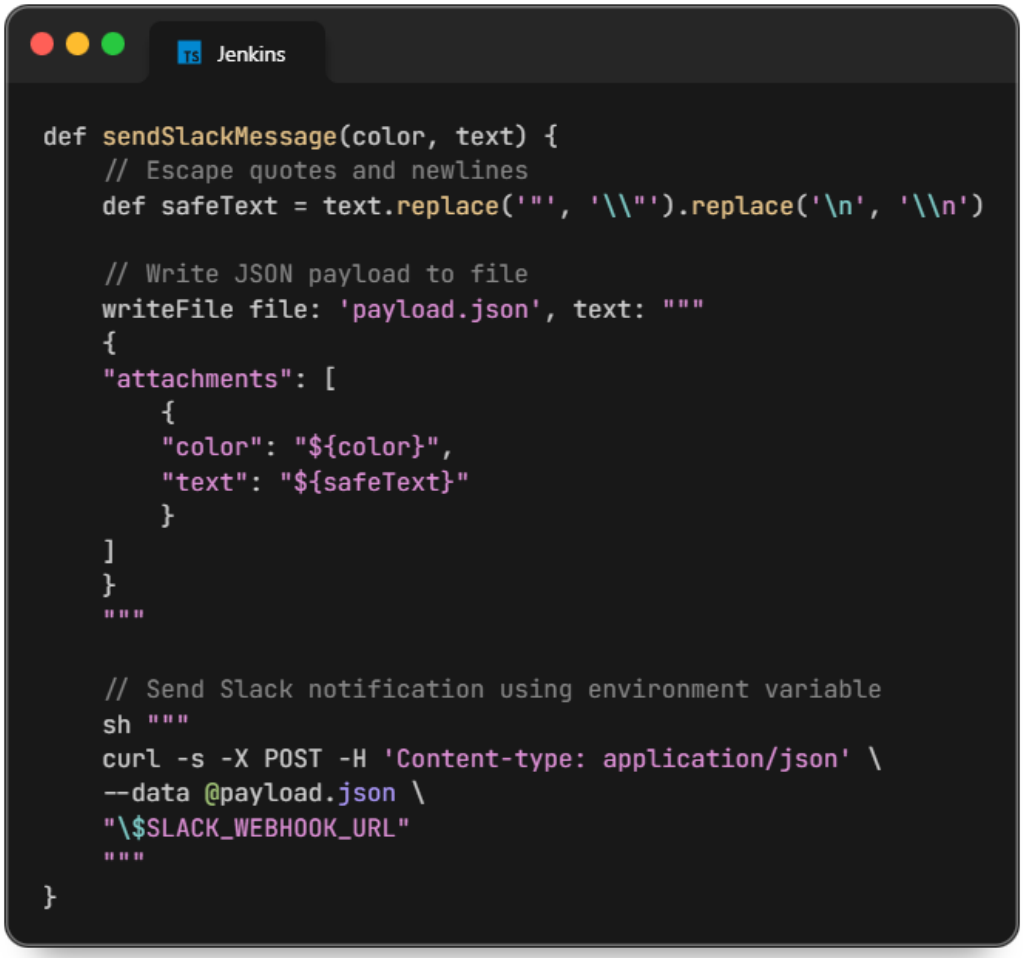

FIGURE 2.5 – Envoi de notifications Slack

2.3 Préparation, construction et tests du projet

. Cette étape regroupe plusieurs actions essentielles pour garantir que le projet est prêt à être déployé :

— Installation des dépendances : toutes les bibliothèques et modules nécessaires sont installés via npm install.

— Construction du projet : le code source est compilé et les fichiers finaux sont générés avec npm run build.

— Exécution des tests automatisés : les tests sont lancés avec npm test pour vérifier la qualité du code. Même en cas d'échec, le pipeline continue afin de ne pas bloquer les étapes suivantes, tout en signalant les problèmes pour analyse.

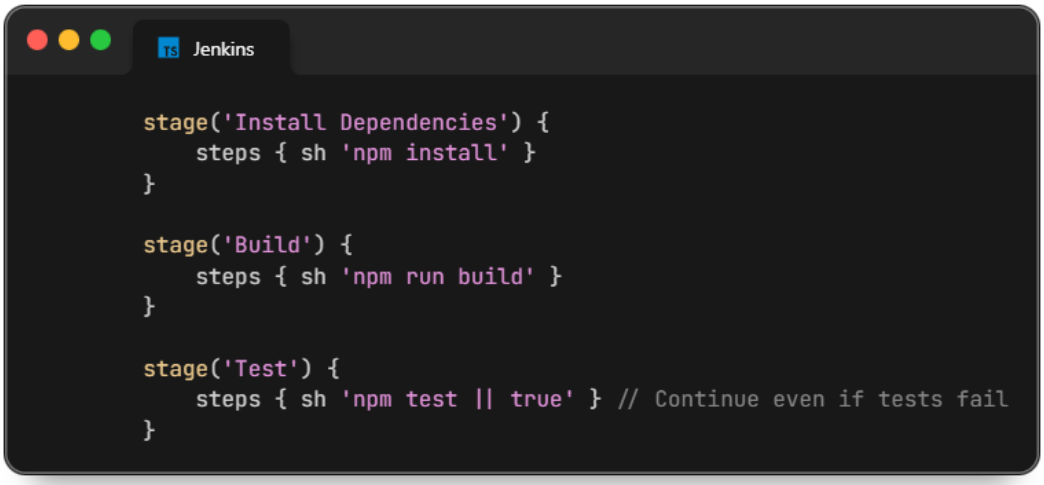

FIGURE 2.6 – Préparation, construction et tests du projet

2.4 Analyse de la qualité du code avec SonarQube

Cette étape utilise SonarQube pour analyser la qualité du code du projet. Le scanner SonarQube examine les fichiers sources, identifie les vulnérabilités, les bugs potentiels et les problèmes de maintenabilité.

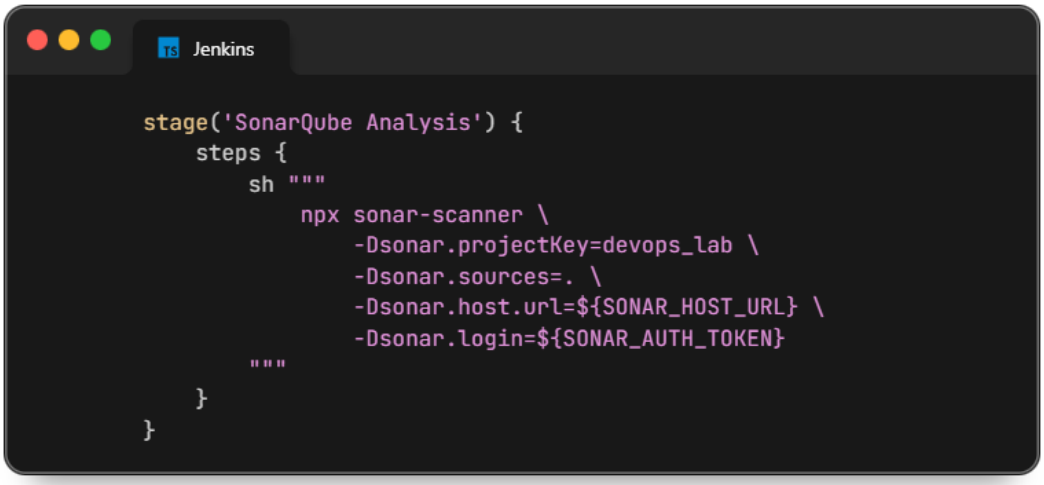

FIGURE 2.7 – SonarQube

L'analyse est réalisée en se connectant au serveur SonarQube grâce à l'URL et au token d'authentification configurés dans Jenkins. Après l'exécution de l'analyse de code, SonarQube fournit un rapport détaillé sur la qualité du projet.

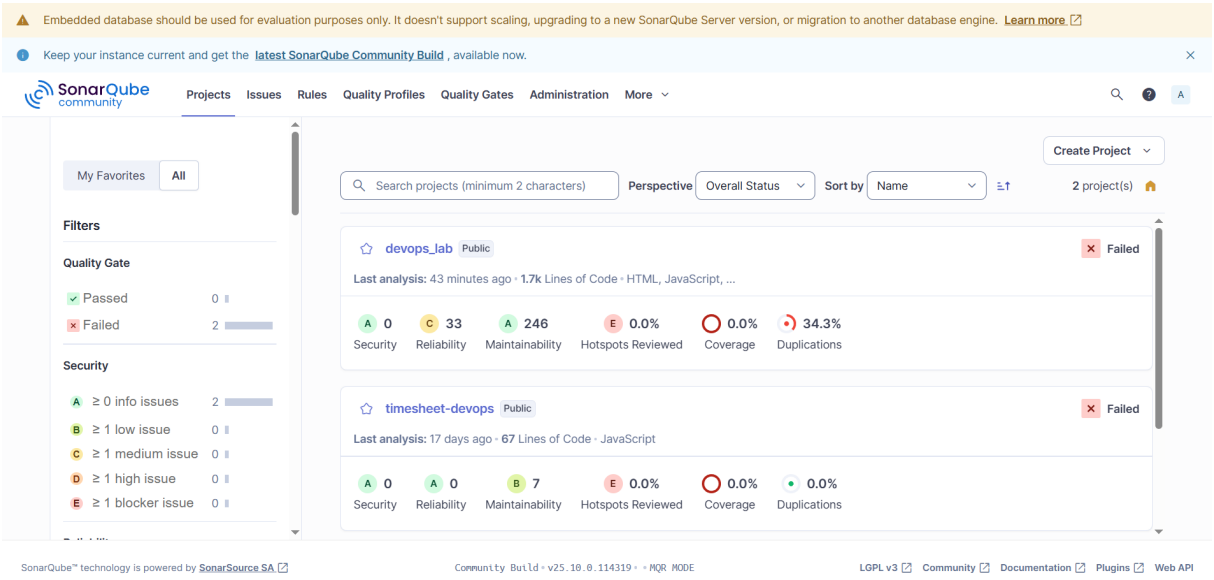

FIGURE 2.8 – SonarQube Report

2.5 Préparation de l'environnement Docker et construction de l'image

Cette étape regroupe deux actions clés pour la conteneurisation de l'application :

— Préparation du réseau Docker : le pipeline supprime d'éventuels conteneurs existants liés à l'application et crée un réseau Docker dédié si celui-ci n'existe pas encore. Cela garantit un environnement propre et isolé pour le déploiement.

— Construction de l'image Docker : l'image Docker de l'application est générée à partir du Dockerfile du projet. Cette image contient toutes les dépendances et configurations nécessaires pour exécuter l'application dans un conteneur.

Jenkins

```
stage('Prepare Docker Network') {
    steps {
        sh "docker rm -f ${APP_CONTAINER} || true"
        sh "docker network inspect ${DOCKER_NETWORK} || docker network create ${DOCKER_NETWORK}"
    }
}

stage('Build Docker Image') {
    steps { sh "docker build -t ${APP_IMAGE} ." }
}
```

FIGURE 2.9 – Préparation de l'environnement Docker

2.6 Déploiement de la stack de monitoring

Cette étape met en place l'ensemble des outils de supervision nécessaires pour observer le comportement de l'application et du système. Le pipeline vérifie d'abord si les conteneurs Prometheus, Grafana et

cAdvisor existent déjà. S'ils ne sont pas présents, ils sont automatiquement créés et lancés dans le réseau Docker dédié.

— Prometheus est déployé avec son fichier de configuration monté depuis le workspace afin de collecter les métriques exposées par les différents services.

— cAdvisor est lancé pour surveiller en temps réel l'activité des conteneurs, notamment l'utilisation CPU, mémoire, disque et réseau.

— Grafana est déployé pour permettre la visualisation et la création de tableaux de bord à partir des métriques collectées.

Jenkins

```
stage('Run Monitoring Stack') {
            steps {
                script {
                    echo '📈 Setting up Prometheus + Grafana + cAdvisor...'

                    def containers = ['prometheus': '9090', 'cadvisor': '5050', 'grafana': '3000']

                    containers.each { name, port →
                        def exists = sh(script: "docker ps -a --filter name=${name} -q", returnStdout: true).trim()
                        if (!exists) {
                            def runCmd = ""
                            if (name == 'prometheus') {
                                runCmd = """
                                    docker run -d --name prometheus \
                                        --network ${DOCKER_NETWORK} \
                                        -p ${PROMETHEUS_PORT}:9090 \
                                        -v ${WORKSPACE}/prometheus.yml:/etc/prometheus/prometheus.yml \
                                        prom/prometheus
                                """
                            } else if (name == 'cadvisor') {
                                runCmd = """
                                    docker run -d --name cadvisor \
                                        --volume=/:/rootfs:ro \
                                        --volume=/var/run/docker.sock:/var/run/docker.sock:ro \
                                        --volume=/var/lib/docker/:/var/lib/docker:ro \
                                        --volume=/sys:/sys:ro \
                                        --network ${DOCKER_NETWORK} \
                                        -p 5050:8080 \
                                        gcr.io/cadvisor/cadvisor:latest
                                """
                            } else if (name == 'grafana') {
                                runCmd = """
                                    docker run -d --name grafana \
                                        --network ${DOCKER_NETWORK} \
                                        -p ${GRAFANA_PORT}:3000 \
                                        grafana/grafana
                                """
                            }
                            sh runCmd
                            echo "✅ ${name} started."
                        } else {
                            echo "✅ ${name} already running."
                        }
                    }
                }
            }
        }
```

FIGURE 2.10 – Stack de monitoring

2.7 Déploiement et vérification du conteneur applicatif

Cette étape déploie le conteneur de l'application et vérifie automatiquement qu'il démarre correctement. Le pipeline supprime d'abord toute instance précédente du conteneur, puis lance la nouvelle version dans le réseau Docker configuré, en exposant le port requis.

Un mécanisme de vérification est ensuite exécuté : le pipeline envoie des requêtes HTTP successives à l'application jusqu'à ce qu'elle réponde avec un code 200 ou 404, indiquant qu'elle est opérationnelle. Si l'application ne répond pas dans le délai imparti, les derniers logs du conteneur sont affichés pour faciliter le diagnostic, et le pipeline est interrompu.

Jenkins

```
stage('Run App Container') {
            steps {
                script {
                    sh "docker rm -f ${APP_CONTAINER} || true"
                    sh """
                        docker run -d --name ${APP_CONTAINER} \
                            --network ${DOCKER_NETWORK} \
                            -p ${APP_PORT}:${APP_PORT} \
                            ${APP_IMAGE}
                    """

                    echo '⏳ Waiting for application to respond...'

                    def maxRetries = 30
                    def sleepTime = 2
                    def appUp = false

                    for (int i = 1; i ⩽ maxRetries; i++) {
                        def response = sh(
                            script: "curl -s -o /dev/null -w '%{http_code}' http://localhost:${APP_PORT}/ || true",
                            returnStdout: true
                        ).trim()

                        if (response == '200' || response == '404') {
                            echo "✅ Application is up!"
                            appUp = true
                            break
                        }

                        echo "Waiting for app... (${i}/${maxRetries})"
                        sleep sleepTime
                    }

                    if (!appUp) {
                        echo "❌ Application did not respond in time."
                        sh "docker logs ${APP_CONTAINER} --tail 50 || true"
                        error("App failed to start")
                    }
                }
            }
        }
```

FIGURE 2.11 – vérification du conteneur applicatif

2.8 Analyse de sécurité du serveur web avec Nikto

Cette étape réalise un scan de sécurité automatisé du serveur web à l'aide de l'outil Nikto. Le pipeline exécute Nikto dans un conteneur Docker afin d'assurer un environnement d'analyse isolé et reproductible. Le scan cible directement le conteneur applicatif en utilisant le réseau Docker du projet.

Jenkins

```
stage('Nikto Web Server Scan') {
    steps {
        script {
            echo '🔍 Running Nikto scan on web server...'
            sh "mkdir -p ${NIKTO_REPORTS}"
            sh """
                docker run --rm -u 0:0 \
                    --network ${DOCKER_NETWORK} \
                    -v ${NIKTO_REPORTS}:/nikto-reports \
                    ghcr.io/sullo/nikto:latest \
                    -h http://${APP_CONTAINER}:${APP_PORT} \
                    -Format html \
                    -o /nikto-reports/nikto-report.html
            """
            archiveArtifacts artifacts: 'nikto-reports/**', allowEmptyArchive: true
            echo '✅ Nikto scan finished, report archived as HTML.'
        }
    }
}
```

FIGURE 2.12 – Analyse de sécurité du serveur web avec Nikto

Les résultats sont générés au format HTML et enregistrés dans un répertoire dédié

Nikto_Report.html

```
myapp / 172.19.0.2 port 4000
Target IP   172.19.0.2
Target hostname myapp
Target Port 4000
HTTP Server
Site Link (Name)    http://myapp:4000/
Site Link (IP)  http://172.19.0.2:4000/
URI /
HTTP Method GET
Description /: Retrieved x-powered-by header: Express.
Test Links  http://myapp:4000/
http://172.19.0.2:4000/
References
URI /
HTTP Method GET
Description /: Suggested security header missing: referrer-policy.
Test Links  http://myapp:4000/
http://172.19.0.2:4000/
References  https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy
URI /
HTTP Method GET
Description /: Suggested security header missing: permissions-policy.
Test Links  http://myapp:4000/
http://172.19.0.2:4000/
References  https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Permissions-Policy
```

FIGURE 2.13 – Nikto Report

2.9 Analyse de sécurité de l'image Docker avec Trivy

Cette étape réalise une analyse approfondie de l'image Docker de l'application à l'aide de Trivy. L'outil est exécuté dans un conteneur Docker afin de garantir un environnement d'analyse stable et isolé. Le scan cible spécifiquement les vulnérabilités de sévérité HIGH et CRITICAL, permettant de détecter rapidement les failles les plus importantes.

Jenkins

```
stage('Trivy Security Scan') {
    steps {
        script {
            echo '🔍 Running Trivy scan on Docker image (pipeline continues)...'
            sh "mkdir -p ${TRIVY_REPORTS}"
            sh """
                # JSON output for automation
                docker run --rm -v /var/run/docker.sock:/var/run/docker.sock \
                    -v ${TRIVY_REPORTS}:/trivy-reports \
                    aquasec/trivy:latest \
                    image --exit-code 0 --severity HIGH,CRITICAL \
                    -f json -o /trivy-reports/trivy-report.json ${APP_IMAGE}

                # Table output for human readability
                docker run --rm -v /var/run/docker.sock:/var/run/docker.sock \
                    -v ${TRIVY_REPORTS}:/trivy-reports \
                    aquasec/trivy:latest \
                    image --exit-code 0 --severity HIGH,CRITICAL \
                    -f table -o /trivy-reports/trivy-report.txt ${APP_IMAGE}
            """
            archiveArtifacts artifacts: 'trivy-reports/**', allowEmptyArchive: true
            echo '✅ Trivy scan finished, reports archived (JSON + table).'
        }
    }
}
```

FIGURE 2.14 – Analyse de sécurité de l'image Docker avec Trivy

Deux formats de rapport sont générés :

— JSON, adapté à l'automatisation et à l'intégration dans d'autres outils,

— Tableau texte, plus lisible pour une consultation humaine.

Les rapports sont stockés dans un dossier dédié puis archivés automatiquement dans Jenkins

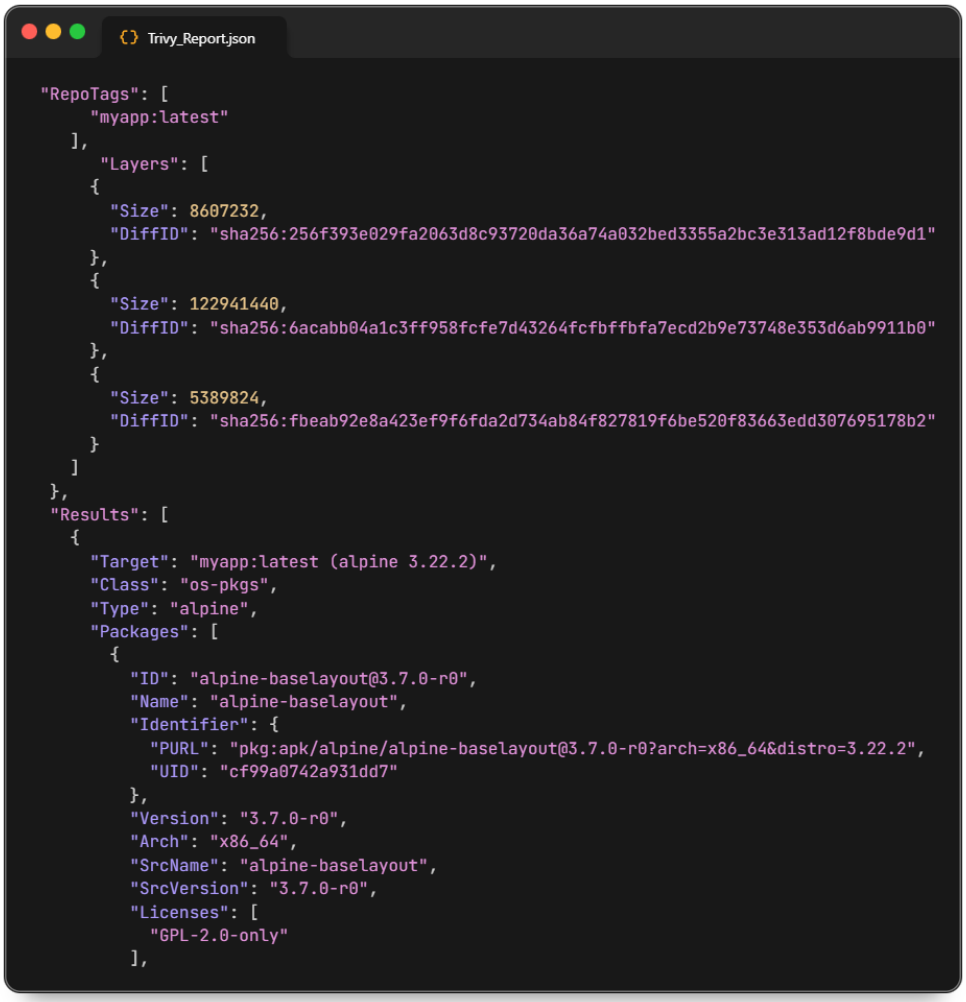

FIGURE 2.15 – Trivy Report

2.10 Notifications et actions post-exécution du pipeline

Cette étape définit les actions effectuées automatiquement à la fin du pipeline, quel que soit son résultat. Trois blocs sont mis en place :

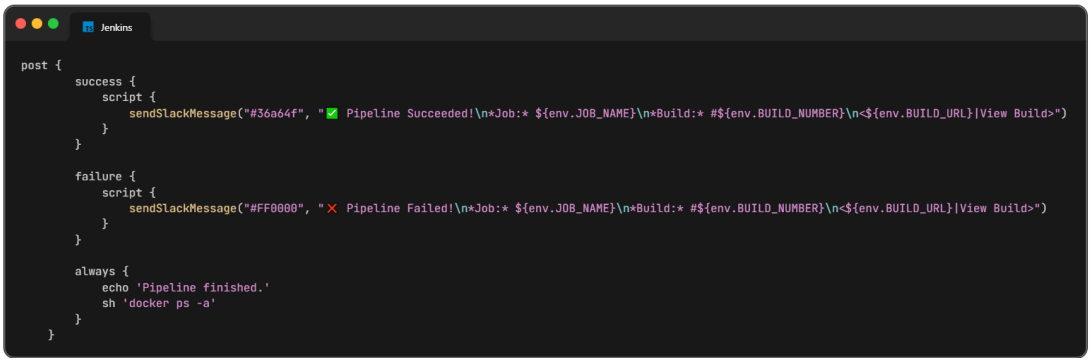

FIGURE 2.16 – Notifications

— Succès du pipeline : un message Slack est envoyé pour signaler que l'exécution s'est terminée cor-

rectement. La notification inclut le nom du job, le numéro du build et un lien direct vers les logs Jenkins.

— Échec du pipeline : en cas d'erreur, une alerte Slack est envoyée avec les mêmes informations, permettant une réaction rapide de l'équipe technique.

— Actions systématiques : indépendamment du statut final, Jenkins affiche un message de fin et liste tous les conteneurs Docker actifs. Cela facilite la vérification de l'environnement et le diagnostic en cas d'anomalie.

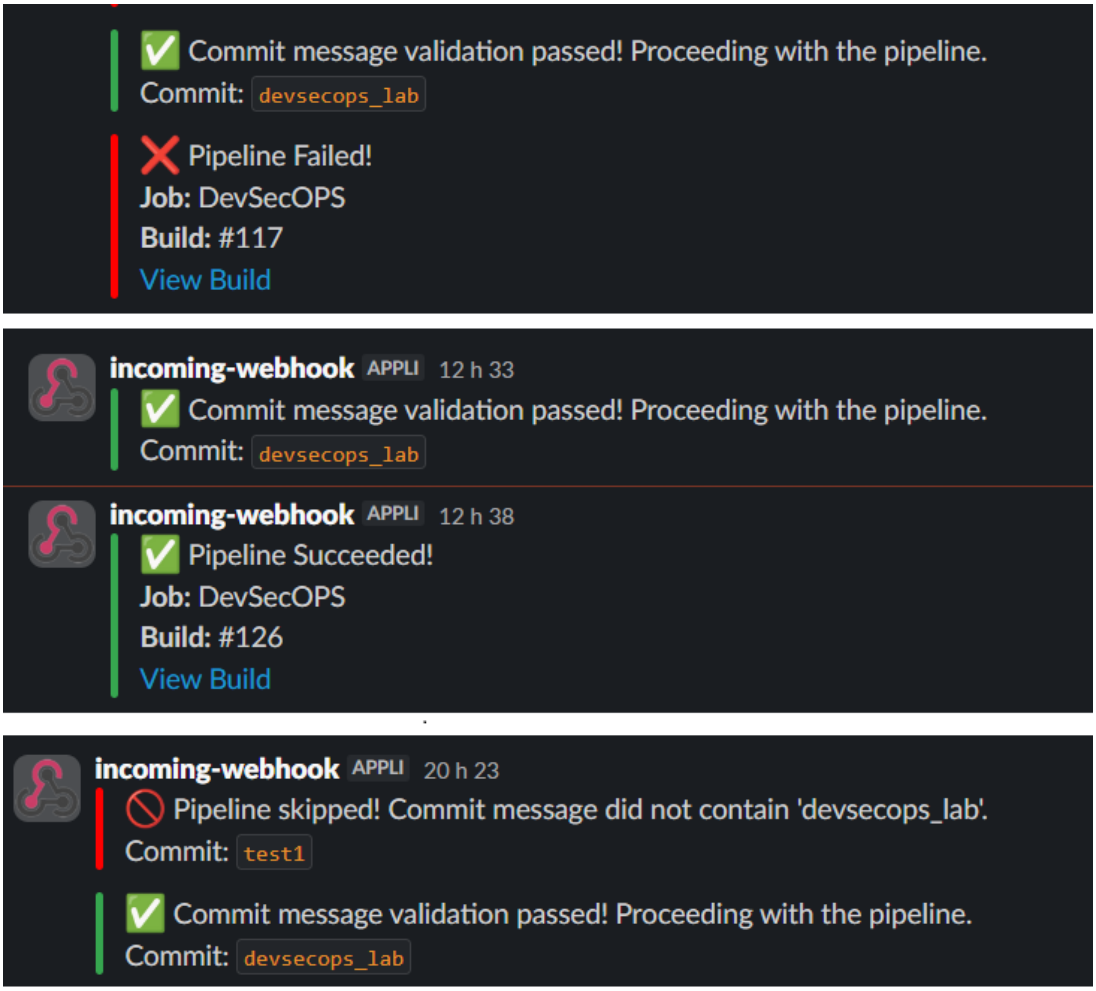

FIGURE 2.17 – Slack

2.11 Déploiement final de l'application

Cette étape marque la phase finale du pipeline, où l'application est déployée dans un conteneur Docker opérationnel. L'image générée précédemment est utilisée pour lancer un nouveau conteneur propre, connecté au réseau Docker dédié et exposé sur le port configuré.

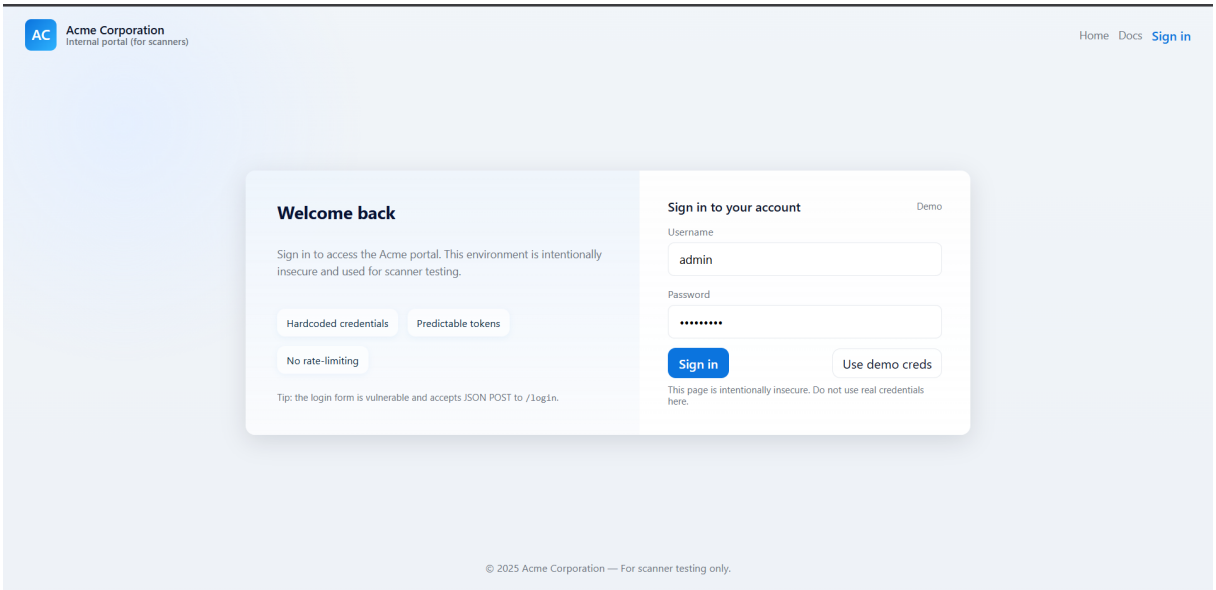

FIGURE 2.18 – application deploy

2.12 Conclusion

Le pipeline CI/CD mis en place intègre l'ensemble des bonnes pratiques DevSecOps en automatisant chaque étape du cycle de vie applicatif : récupération du code, validation des commits, installation des dépendances, construction, tests, analyse de sécurité, conteneurisation, surveillance et déploiement. Les contrôles automatisés via SonarQube, Trivy, Nikto et le monitoring Prometheus/Grafana renforcent la sécurité et la visibilité opérationnelle dès les premières phases du développement.

L'intégration de notifications Slack, l'isolation des services via Docker et la mise en place d'un déploiement contrôlé garantissent un processus fiable, reproductible et conforme aux exigences de qualité et de sécurité. Ce pipeline constitue ainsi une base solide pour une démarche DevSecOps mature, permettant d'accélérer la livraison de fonctionnalités tout en assurant un niveau élevé de sécurité et de robustesse.