Devops QA

DevSecOps com Trivy em Pipelines Jenkins

25 de novembro de 2025

Devops QA

DevSecOps com Trivy em Pipelines Jenkins

25 de novembro de 2025

Sumário

1 Introdução

Este documento relata sobre DevSecOps aplicada a uma aplicação Java com Spring Boot, empacotada em Docker e integrada a pipelines de *Continuous Integration/Continuous Delivery* (CI/CD) usando Jenkins.

A ideia central é mostrar como evoluir de um fluxo tradicional de DevOps para um fluxo de **DevSecOps**, incorporando varreduras de segurança automatizadas com a ferramenta **Trivy** em diferentes pontos do pipeline:

- varredura do *filesystem* do repositório (`trivy fs .`);
- varredura da imagem de container gerada (`trivy image ...`);
- uso de arquivos de configuração como `.trivyignore` e `.trivy.yaml`;
- integração com `Jenkinsfile` para criar *gates* de segurança antes do deploy.

2 De DevOps a DevSecOps

Em um fluxo de DevOps clássico, o pipeline costuma envolver:

1. **Build** da aplicação (por exemplo, `mvn clean package`);
2. **Testes automatizados** (unidade, integração);
3. **Build da imagem Docker**;
4. **Deploy** (manual ou automatizado) em ambientes de *staging* e produção.

Ao migrar para **DevSecOps**, adicionamos um eixo de *segurança desde o início* (*shift-left security*). Em linhas gerais, o fluxo passa a ser:

1. build e testes com Maven;
2. build da imagem Docker (sem ainda subir para o registro);
3. **Trivy — varredura do código / dependências** (`trivy fs .`);
4. **Trivy — varredura da imagem** (`trivy image nome-da-imagem`);
5. se não houver vulnerabilidades críticas:
    a) push da imagem para o registro (Docker Hub, GitLab Registry etc.);
    b) deploy em staging/produção, por exemplo via `docker-compose` ou Kubernetes.

O ponto chave do DevSecOps é transformar a segurança em critério de aprovação do pipeline: se forem identificadas vulnerabilidades de severidade *HIGH* ou *CRITICAL*, o *build* é marcado como *FAIL* e o deploy é bloqueado até que as correções sejam realizadas.

3 Visão Geral do fluxo CI/CD seguro

Considere o seguinte fluxo simplificado em Jenkins para uma aplicação Java/Spring Boot:

1. **Stage: Checkout** — obtém o código-fonte do repositório;
2. **Stage: Build & Test** — roda `mvn clean verify`;
3. **Stage: Build Docker Image** — constrói a imagem com `docker build`;
4. **Stage: Trivy FS Scan** — roda `trivy fs .` sobre o repositório;
5. **Stage: Trivy Image Scan** — roda `trivy image nome-da-imagem`;
6. **Stage: Push** — envia a imagem aprovada para o registro;
7. **Stage: Deploy** — faz o deploy em staging/produção.

Os estágios de Trivy devem ser configurados com `--exit-code 1` para que o pipeline seja interrompido quando forem encontradas vulnerabilidades relevantes.

4 Integração do Trivy no pipeline Jenkins

4.1 Instalação do Trivy no agente Jenkins

No servidor ou agente onde o Jenkins executa os pipelines, é necessário instalar o Trivy e deixá-lo disponível no `PATH`. Em um ambiente Linux, isso pode ser feito, por exemplo, com:

Listing 4.1 – Instalação do Trivy em Linux (exemplo)

```
sudo apt-get update
sudo apt-get install -y wget
wget https://github.com/aquasecurity/trivy/releases/latest/
   download/trivy_Linux-64bit.tar.gz
tar zxvf trivy_Linux-64bit.tar.gz
sudo mv trivy /usr/local/bin/
trivy --version
```

Uma vez instalado, o Jenkins pode chamar o Trivy diretamente nas etapas do pipeline usando `sh` (Linux) ou `bat` (Windows).

4.2 Pipeline Jenkins completo com Trivy (CI)

O trecho a seguir mostra um exemplo de pipeline declarativo em Jenkins, que realiza:

- build e testes com Maven;
- build da imagem Docker;
- varredura com Trivy no filesystem;
- varredura com Trivy na imagem;
- push da imagem para o Docker Hub.

Listing 4.2 – Pipeline Jenkins com Trivy para CI

```
pipeline {
  agent any

  environment {
```

```
5       REGISTRY   = 'andprof'                  // usuario Docker Hub
6       IMAGE_NAME = 'ac2_ca'
7       IMAGE_TAG  = "build-${env.BUILD_NUMBER}"
8       FULL_IMAGE = "${REGISTRY}/${IMAGE_NAME}:${IMAGE_TAG}"
9     }
10
11    stages {
12      stage('Checkout') {
13        steps {
14          checkout scm
15        }
16      }
17
18      stage('Build & Test (Maven)') {
19        steps {
20          bat 'mvn clean verify -DskipTests=false'
21        }
22      }
23
24      stage('Build Docker Image') {
25        steps {
26          bat "docker build -t ${FULL_IMAGE} ."
27        }
28      }
29
30      stage('Trivy Scan - Filesystem') {
31        steps {
32          bat '''
33            trivy fs --exit-code 1 --severity HIGH,CRITICAL ^
34              --format table --output trivy-fs-report.txt .
35          '''
36          archiveArtifacts artifacts: 'trivy-fs-report.txt',
37                           onlyIfSuccessful: false
38        }
39      }
40
41      stage('Trivy Scan - Image') {
42        steps {
43          bat """
44            trivy image --exit-code 1 --severity HIGH,CRITICAL ^
45              --format table --output trivy-image-report.txt ${
```

```
                FULL_IMAGE}
46          """
47          archiveArtifacts artifacts: 'trivy-image-report.txt',
48                           onlyIfSuccessful: false
49        }
50      }
51
52      stage('Docker Login & Push') {
53        when {
54          expression { currentBuild.resultIsBetterOrEqualTo('
                SUCCESS') }
55        }
56        steps {
57          withCredentials([usernamePassword(credentialsId: '
                DOCKERHUB_CRED',
58                                            usernameVariable: '
                                                 DOCKER_USER',
59                                            passwordVariable: '
                                                 DOCKER_PASS')]) {
60            bat "docker login -u %DOCKER_USER% -p %DOCKER_PASS%"
61            bat "docker push ${FULL_IMAGE}"
62            bat "docker tag ${FULL_IMAGE} ${REGISTRY}/${IMAGE_NAME
                   }:latest"
63            bat "docker push ${REGISTRY}/${IMAGE_NAME}:latest"
64          }
65        }
66      }
67    }
68
69    post {
70      always {
71        echo 'Pipeline CI finalizado (com DevSecOps via Trivy).'
72      }
73    }
74  }
```

Note que:

- os estágios de Trivy usam `--exit-code 1` e `--severity HIGH,CRITICAL`;
- os relatórios (`trivy-fs-report.txt` e `trivy-image-report.txt`) são arquivados como artefatos do Jenkins.

5 Pipelines de produção e homologação com Trivy

Se você já possui pipelines específicos para produção (`Jenkinsfile.prod`) e homologação (`Jenkinsfile.staging`), é possível incluir um estágio de Trivy antes da subida dos containers.

Por exemplo, em produção:

Listing 5.1 – Exemplo simplificado de Jenkinsfile.prod com Trivy

```
pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('Security Scan - Image (Trivy)') {
      steps {
        bat '''
          trivy image --exit-code 1 --severity HIGH,CRITICAL ^
            --format table --output trivy-prod-image-report.txt
               andprof/ac2_ca:latest
        '''
        archiveArtifacts artifacts: 'trivy-prod-image-report.txt
           ',
                         onlyIfSuccessful: false
      }
    }

    stage('Start container') {
      steps {
        echo 'Starting container from Docker Hub...'
        bat 'docker-compose -f docker-compose.prod.yml pull'
        bat 'docker-compose -f docker-compose.prod.yml up -d --no
           -color'
```

```
27          sleep time: 60, unit: 'SECONDS'
28          bat 'docker-compose -f docker-compose.prod.yml logs'
29          bat 'docker-compose -f docker-compose.prod.yml ps'
30        }
31      }
32
33      stage('Run tests against the container') {
34        steps {
35          bat 'curl http://localhost:8585 || echo "Service not
                responding"'
36        }
37      }
38    }
39
40    post {
41      always {
42        echo 'Pipeline completed'
43      }
44    }
45  }
```

A mesma abordagem pode ser usada em pipelines de *staging*, mudando o arquivo `docker-compose` e as portas conforme necessário.

6 Hardening da imagem Docker

A seguir, apresentamos um comparativo entre um *Dockerfile* básico e uma versão levemente *hardened* para uma aplicação Java/Spring Boot. Hardening é o processo de reduzir a superfície de ataque de um sistema, aplicação, servidor ou imagem de container, aplicando configurações que aumentam sua segurança e minimizam vulnerabilidades exploráveis. Isso inclui remover serviços desnecessários, restringir permissões, aplicar políticas de acesso mais rígidas, usar usuários não privilegiados, atualizar dependências, configurar firewalls e validar configurações seguras em todo o ambiente. O objetivo é tornar o sistema mais resistente a ataques, impedindo que falhas simples se tornem brechas críticas.

6.1 Dockerfile básico

Listing 6.1 – Dockerfile básico com OpenJDK 17

```
1 FROM openjdk:17
2
3 WORKDIR /ac2_ca
4 COPY target/*.jar /ac2_ca/ac2_ca-0.0.1-SNAPSHOT.jar
5 EXPOSE 8585
6 CMD ["java", "-jar", "ac2_ca-0.0.1-SNAPSHOT.jar"]
```

6.2 Dockerfile com ajustes de segurança

Listing 6.2 – Dockerfile com hardening e usuario nao root

```
1  FROM eclipse-temurin:17-jre-alpine
2
3  WORKDIR /app
4  COPY target/*.jar app.jar
5
6  # Cria usuario nao root
7  RUN addgroup -S spring && adduser -S spring -G spring
8  USER spring
9
10 EXPOSE 8585
11
12 ENTRYPOINT ["java","-jar","app.jar"]
```

As principais melhorias são:

- uso de uma imagem base mais enxuta (`alpine`), reduzindo a superfície de ataque;
- execução da aplicação com um usuário não root.

O Trivy irá refletir essas mudanças na forma de menos vulnerabilidades reportadas na imagem final.

7 Arquivos de configuração do Trivy

7.1 .trivyignore

O arquivo `.trivyignore` pode ser usado para ignorar vulnerabilidades específicas, por exemplo quando já foram analisadas e mitigadas por outros controles de segurança:

Listing 7.1 – Exemplo de arquivo .trivyignore

```
# Ignora CVEs especificos ja analisados
CVE-2023-12345
CVE-2022-98765
```

7.2 .trivy.yaml

O arquivo `.trivy.yaml` (ou `.trivy.yml`) permite definir parâmetros padrão de execução, como severidade mínima, tipos de vulnerabilidade e tempo limite:

Listing 7.2 – Exemplo de arquivo .trivy.yaml

```
severity:
  - CRITICAL
  - HIGH
vuln-type:
  - os
  - library
ignore-unfixed: true
format: table
timeout: 5m
```

A partir desse arquivo, chamadas como `trivy fs .` e `trivy image ...` já herdam essas configurações, simplificando o uso em scripts e pipelines.

8 Conclusão

Este documento mostrou uma forma prática de evoluir de um fluxo de DevOps tradicional para um fluxo de DevSecOps, integrando a ferramenta Trivy em pipelines Jenkins para:

- varrer o código-fonte e as dependências;
- analisar imagens Docker antes do push e do deploy;
- criar *gates* de segurança com base em severidades de vulnerabilidade;
- incentivar boas práticas de *hardening* de imagens.

Você pode estender esta apostila incluindo:

- seções sobre testes de segurança de APIs;
- exemplos de integração com Kubernetes e *Helm*;
- estudos de caso reais envolvendo aplicações Java/Spring Boot e pipelines de CI/CD.