Automated Docker Image Security Scanning Pipeline

Introduction: Embracing DevSecOps

In modern software development, the speed of delivery is paramount. Continuous Integration and Continuous Deployment (CI/CD) pipelines, powered by tools like Jenkins and Docker, have become the backbone of this rapid delivery model. However, speed without security can lead to disaster. The rise of containerization, while solving many dependency and environment issues, introduces a new layer of security concerns. Docker images can contain vulnerabilities inherited from base images, outdated systelibraries, or insecure application dependencies.

This is where the principle of **DevSecOps** comes into play—integrating security practices directly into the DevOps lifecycle. The goal is to "shift left," addressing security concerns as early as possible in the development process rather than treating security as an afterthought or a final gate before production. An automated Docker image security scanning pipeline is a cornerstone of a mature DevSecOps strategy.

> *This project details how to build a Jenkins pipeline that not only builds Docker images but also automatically scans them for security vulnerabilities. By embedding a security scanner directly into the CI workflow, we create an automated quality gate that prevents insecure images from ever reaching a container registry or a production environment.*

The Core Value Proposition

Integrating automated security scanning into your Docker build pipeline provides immense value, transforming security from a manual, time-consuming task into a seamless, automated part of your development culture.

- **Early Vulnerability Detection:** Instead of discovering a critical vulnerability in a production system, you catch it the moment a new dependency is added or a base image becomes outdated. This drastically reduces the cost and complexity of remediation.
- **Continuous Enforcement of Security Policies:** Automation ensures that every single build is scrutinized against your organization's security standards. No image gets a free pass, enforcing a consistent security posture across all projects.
- **Developer Empowerment and Education:** When a pipeline fails due to a security issue, developers receive immediate, actionable feedback. This encourages them to use updated base images, keep dependencies in check, and become more security-conscious in their daily work.
- **Reduced Risk and Increased Trust:** By automating these checks, you build a more secure software supply chain. Stakeholders, from security teams to end-users, can have greater confidence that the deployed applications have been vetted for known vulnerabilities.
- **Operational Efficiency:** Security teams are freed from the burden of manually scanning every image. Jenkins handles the heavy lifting of scanning, reporting, and blocking insecure deployments, allowing security experts to focus on more complex threats.

Goal:

Use Jenkins to automatically build, scan, and push Docker images — blocking insecure ones before deployment.

We'll use **Trivy** (a popular open-source vulnerability scanner).

Step-by-Step Implementation

1. Prerequisites Setup

🖥️ Local or Cloud Environment

You can do this on:

- Local machine (Windows/Linux with Docker Desktop)
- Or cloud VM (e.g., AWS EC2 Linux)

Install Docker

```
sudo yum install docker -y
```

```
# Start Docker
sudo systemctl enable docker
sudo systemctl start docker

# Add Jenkins user to Docker group
sudo usermod -aG docker jenkins
```

Install Jenkins

```
sudo yum update –y
sudo wget -O /etc/yum.repos.d/jenkins.repo \
    https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.
sudo yum upgrade
sudo yum install java-17-amazon-corretto -y
sudo yum install jenkins -y
sudo systemctl enable jenkins
sudo systemctl start jenkins
sudo systemctl status jenkins
```

Then open Jenkins in your browser:
👉 `http://<your-server-ip>:8080`

Set up:

- Install **suggested plugins**

- Create admin user
- Add **Docker Pipeline Plugin**

Choosing a security scanning tool

The first step is to select a tool to perform the vulnerability scanning. The market offers several excellent open-source options. The reference material highlights Trivy, Clair, and Anchore. Let's explore these in more detail to make an informed decision.

Comparison of Popular Scanners

Feature	Trivy	Clair	Anchore Engine
Primary Focus	Simplicity, speed, and comprehensive vulnerability detection.	Static analysis of vulnerabilities in application containers. API-driven.	Deep image inspection, policy-based compliance, and vulnerability scanning.
Ease of Use	Extremely high. It's a single binary with no server or database dependencies required for basic use.	Moderate. Requires a running server and a PostgreSQL database to store vulnerability data.	Moderate to complex. Requires a multi-service architecture (API, catalog, policy engine, etc.).
Scanning Targets	OS packages (Alpine, RHEL, CentOS, Debian, Ubuntu, etc.), application dependencies (npm, pip, Maven, Go, etc.), IaC files (Terraform, Dockerfile), and secrets.	Primarily OS packages. Application dependency scanning is less comprehensive than Trivy.	OS packages, application dependencies, file contents, and configuration files. Can generate a detailed SBOM.
Vulnerability DB	Maintains its own vulnerability database, which is updated	Ingests vulnerability data from various sources (NVD, Debian Security Bug	Aggregates data from multiple sources and

Feature	Trivy	Clair	Anchore Engine
	frequently. Downloads a local copy on first run.	Tracker, etc.) into its own database.	provides its own curated feed.
CI/CD Integration	Excellent. The command-line interface (CLI) is designed for CI. Can fail a build based on severity using an exit code.	Good. Typically integrated via its API. Requires a client (like `clair-scanner`) to interact with the Clair server.	Good. Integrated via its API or the `anchore-cli`. The policy engine is powerful for CI gates.
Output Formats	Table (default), JSON, SARIF, CycloneDX, SPDX. Very flexible for reporting.	JSON. Less flexible out-of-the-box.	JSON, and custom reports via its policy evaluation system.

Recommendation: Trivy

For most use cases, especially when starting out, **Trivy** is the recommended choice. Its combination of simplicity, speed, and comprehensive scanning capabilities makes it ideal for seamless integration into a Jenkins pipeline. You can get meaningful results in minutes without the overhead of setting up and maintaining a complex server architecture.

Why Trivy stands out for this project:

- **Zero-dependency CLI:** You can run it directly in a `sh` step in your Jenkinsfile.
- **Fast Scans:** Keeps the CI pipeline execution time low.
- **Built-in CI Failure Logic:** The `--exit-code` flag is perfect for creating a security gate.
- **Rich Output Formats:** Supports JSON for machine processing and SARIF for integration with modern code analysis tools and Jenkins plugins.

Installation

You can install Trivy on your Jenkins agent or, even better, run it from its official Docker image. The latter approach avoids polluting your agent with extra tools.

To install on an amazon linux server:

```
# Step 1: Go to /opt or home directory
cd /opt

# Step 2: Download the latest Trivy RPM package
sudo wget https://github.com/aquasecurity/trivy/releases/download/v0.50

# Step 3: Install it using rpm with sudo privileges
sudo rpm -ivh trivy_0.50.1_Linux-64bit.rpm
```

```
# You should see output like:
Preparing...                          #################################
Updating / installing...
   1:trivy-0.50.1-1                   #################################
```

```
trivy --version

# You can now test:
trivy image hello-world
```

However, for this guide, we will focus on the container-based approach, which is cleaner and more portable.

Create a Sample App + Dockerfile

Make a simple app.py for testing:

```
# --- app.py ---
import os
from flask import Flask

app = Flask(__name__)

@app.route("/")
```

```
def main():
    return "Welcome to Batch11!"

@app.route('/how are you')
def hello():
    return 'I am good, how about you?'

if __name__ == "__main__":
    app.run()
```

```
# --- Dockerfile ---
FROM python:3.9-slim
RUN pip install flask
WORKDIR /opt/python
COPY app.py .
ENV FLASK_APP=app.py
EXPOSE 8090
CMD ["flask", "run", "--host=0.0.0.0", "--port=8090"]
```

Push this code to your GitHub repo (e.g., `trivy-pipeline`).

Configure jenkins credentials

In Jenkins:

- **Go to Manage Jenkins → Credentials → Global → Add Credentials**
- **Add your GitHub credentials (username + token)**
- **Add your Docker Hub credentials (for pushing images)**

Create Jenkins Pipeline Job

Go to Jenkins → "New Item" → **Pipeline**
Name: `docker-security-pipeline`

Then in the **Pipeline Script** box, paste this:

```
pipeline {
    agent any

    environment {
        DOCKER_IMAGE = "kadivetikavya/app:${BUILD_ID}"
    }

    options {
        timeout(time: 60, unit: 'MINUTES')
        timestamps()
    }

    stages {

        stage('Clone Repository') {
            steps {
                git branch: 'main', url: 'https://github.com/kadivetika
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    def dockerImage = docker.build("${DOCKER_IMAGE}")
                }
            }
        }

        stage('Scan Docker Image') {
            steps {
                script {
                    sh '''
                        mkdir -p /var/lib/jenkins/.cache/trivy
                        trivy image --cache-dir /var/lib/jenkins/.cache
                                    --timeout 5m \
                                    --exit-code 1 \
                                    --severity HIGH,CRITICAL \
                                    ${DOCKER_IMAGE} || true

                        trivy image --cache-dir /var/lib/jenkins/.cache
                                    --format table \
                                    -o trivy-report.txt \
                                    ${DOCKER_IMAGE} || true
                    '''
                }
            }
        }

        stage('Push to Docker Hub') {
            when {
                expression {
                    // Push only if no CRITICAL vulnerabilities found
                    sh(script: "trivy image --exit-code 1 --severity CR
                }
            }
            steps {
                withCredentials([usernamePassword(credentialsId: 'kavya
                    sh '''
                        echo $PASS | docker login -u $USER --password-s
                        docker push ${DOCKER_IMAGE}
                    '''
                }
            }
        }
    }

    post {
        always {
            archiveArtifacts artifacts: 'trivy-report.txt', fingerprint
        }
    }
}
```

Run the Pipeline

Click Build Now 🧠

Watch the stages:

1. **Clone code**
2. **Build Docker image**
3. **Scan image (Trivy)**

4. **Push to Docker Hub (only if secure)**

If Trivy detects critical vulnerabilities, the pipeline fails before deployment

Verify

After success:

- **Check Docker Hub → Image uploaded**
- **Check Jenkins “Artifacts” → `trivy-report.txt` (contains vulnerability scan summary)**

> By implementing this automated security scanning pipeline, you establish a robust DevSecOps practice. Jenkins, combined with powerful tools like Trivy, ensures that every Docker image is scrutinized, helping your team shift security left, catch ild