International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 02 | Feb - 2025 SJIF Rating: 8.448 ISSN: 2582-3930

Docker Like a Pro: Essential Practices for Secure and Scalable Containers

Author(s):

Kriti Khattar
Docker Like a Pro

Abstract

Adopting best practices in project development is crucial for ensuring security, efficiency, and sustainability. Key practices include minimizing security risks by limiting access permissions, optimizing container images for faster deployments, and automating processes to reduce manual intervention. Additionally, effective configuration management, container health checks, and seamless integration of security tools within CI/CD pipelines help maintain system reliability and code quality. By leveraging cross-account access, choosing secure base images, and automating container restarts, projects can maintain operational stability while minimizing downtime. Overall, these practices foster a secure, scalable, and well-maintained environment for application development and deployment.

Keywords: Docker Security Practices, Principle of Least Privilege, Multi-Stage Docker file, Docker Image Optimization, Configuration Management, Docker Tagging, Container Health Check, Base Image Selection, Cross-Account Access, CI/CD Pipeline Security, SonarQube, Prisma Cloud, Container Automation

Introduction

In DevOps, containerization has revolutionized software deployment by enabling consistency, scalability, and faster release cycles. However, without proper security measures and best practices, containers can introduce vulnerabilities and inefficiencies. This paper provides a comprehensive guide to Docker security and deployment best practices that help DevOps teams build, deploy, and manage containerized applications efficiently while ensuring security, performance, and resilience.

Few of the practices that we can implement in our project are:

1.Apply the Principle of Least Privilege for User Access

By default, Docker file does not specify a user, it uses the root user. The functionality can run fine without root permissions as well.

This can cause security issue since the container starts on the host, it potentially has root access on the Docker host.

If an attacker exploits a vulnerability in the application, they can gain control not only over the container but potentially over the underlying server and its processes as well. This makes the entire server more susceptible to being compromised.

How can we avoid this?

```
# Add acp user
RUN useradd -m acp --system; \
    passwd -l acp; \
    mkdir /tmp/src; \
    chown acp:acp /tmp/src

# Change the User
USER acp

#Copy source code
COPY --chown=acp:acp src /tmp/src
COPY --chown=acp:acp pom.xml /tmp
```

Create a new user and group acp. Then set a password and change the ownership and

permission as required. This helps to switch to that user and can perform our task with ease.

2.MultiStage Docker file Breakdown

We can use docker file in stages.

i. **Stage1**

a. **Build the environment in isolation:** This ensures that the final runtime image does not contain unnecessary build tools or dependencies.

```
# Stage 1:
# Build the socurce code using maven
#-----------------------------------------------
FROM [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-common:maven-3.8.5-openjdk-17 AS java-compiler
```

b. **Reusability:** The build stage can be reused in different pipelines if required.

```
COPY --chown=acp:acp --from=java-compiler /tmp/target/acp-pii-handler.jar /opt/acp/acp-pii-handler.jar
COPY --chown=acp:acp --from=java-compiler /tmp/target/config /opt/acp/config
```

In stage1 create a runtime image and name it as java-compiler & reused it in later stage.

c. **Security**: The final image is smaller in size and does not include build tools. Thus, reducing the chance of attack.

ii. **Stage2:**

a. **Minimal Image Size**: Create a runtime image of very small by only including the necessary runtime dependencies and the compiled application. Reducing the image size results in faster startup times and lower resource consumption.

```
# Stage 2:
# Build image with minimum resources
#-----------------------------------------------
#docker hub image eclipse-temurin:17.0.10_7-jdk
FROM [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-common:eclipse-temurin-17-0-10-7-jdk as java-runtime
```

3. Manage Configuration Changes During Deployment

As a best practice, Docker says **"Build once, Run anywhere concept".**

This can be implemented by storing our docker image in ECR and then reusing this image in any environment in future. This has been implemented in our buildspec.yml

```
# Push docker images to AWS ECR repo
- echo Pushing the Docker image...
- docker push [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-pii-handler:latest
- docker push [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-pii-handler:"$CODEBUILD_BUILD_NUMBER"_"$temp"
```

Handle other dependencies in our deployment and not in docker file which makes it readily usable in any environment.

This can been implemented in appspec.yml

```
ApplicationStart:
  - location: scripts/update_prop.sh
    timeout: 60
    runas: root
```

4.Docker tag using Commit ID

Generally, the container tags build by code build contains build number only. We have implemented a tag that mentions the commit id along with build number. Easier roll back since it helps find the commit user and commit history.

We have implemented in our buildspec.yml to accommodate this.

```
var=$(git rev-parse HEAD 2>&1)
echo $var
temp=$(echo "$var" | cut -c 1-8)
echo $temp
docker build -t [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-pii-handler:latest .
docker tag [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-pii-handler:latest [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-pii-handler:"$CODEBUILD_BUILD_NUMBER"_"$temp"
```

5. Health check for container

Docker health checks monitor the application inside the container using the HEALTHCHECK instruction, ensuring it runs as expected. A monitoring script periodically inspects the container's health status using docker inspect. If the container is marked as "unhealthy," the script automatically sends a

notification to a configured Microsoft Teams channel, enabling real-time alerts and prompt issue resolution.

```
HEALTHCHECK --interval=35s --timeout=30s --retries=3 CMD curl --fail 'http://localhost:8000/health' || exit 1
```

Docker runs the health check every 35 seconds (--interval=35s) and allows up to 30 seconds for each check to complete (--timeout=30s). If the check fails, Docker will retry it 3 times (--retries=3) before marking the container as unhealthy. The command CMD curl --fail 'http://localhost:8000/health' || exit 1 sends an HTTP request to the /health endpoint of the application inside the container. If the request fails or returns a non-successful status code, the command exits with a non-zero code, indicating a failed health check.

6. *Choosing the Most Efficient Base Image for Your Container*

When choosing between a JRE (Java Runtime Environment) and JDK (Java Development Kit) for a Docker container, the decision hinges on the container's use case. If the container's main task is to **run Java applications** and you don't need to compile or develop Java code, the **JRE Alpine version** (e.g., openjdk:17-jre-alpine) is the better option. The JRE only includes the essential runtime components, making the image **lighter** and more **efficient** in terms of size and resource consumption. Conversely, if your container needs to **compile Java code** or use development tools like the Java compiler, a **JDK Alpine version** (e.g., openjdk:17-alpine) is required. While **JDK images** are more **heavyweight** due to the inclusion of development tools, they are necessary for compiling and developing Java applications. For most production environments, where you only need to execute Java applications, the **JRE Alpine** image is preferred to keep the container **smaller** and **optimized**.

Sample image names:

JRE Alpine: openjdk:17-jre-alpine

JDK Alpine: openjdk:17-alpine

7. Cross-Account Access to AWS Code Commit Repository in Jenkins Pipeline

In this process, the pipeline in one account uses AWS credentials to access a Code Commit repository in another account. For cross-account access, these credentials must be configured with permissions to assume a role in the target account, granting access to the repository. The pipeline begins by setting up the **AWS CLI with the appropriate credentials**, allowing interaction with AWS resources. Next, Git is configured to use the **Code Commit credential helper** for secure authentication without storing credentials manually. The pipeline then executes a git clone command to retrieve the repository from Code Commit. This process ensures secure, temporary authentication and enables seamless access to repositories across accounts.

8. Importance of Docker Hub References in Docker file Comments

Adding Docker Hub references in comments within a Docker file provides several key benefits. First, it offers clear **documentation**, allowing developers to easily access additional information about the base image, including details, versioning, and usage instructions on Docker Hub. Second, it helps with **version control**, enabling developers to track the latest updates and security patches for the base image by referencing its official Docker Hub page. Third, it promotes **collaboration** within teams, as it helps others understand the reasoning behind choosing

specific base images, ensuring consistency across environments. Finally, using official Docker Hub images ensures **compliance and best practices**, as these images are regularly updated and maintained, making them secure and trusted. This approach enhances the overall clarity, maintainability, and security of Docker files.

```
# Stage 1:
# Install Dependencies using pip install
# Docker Image can be found here: https://hub.docker.com/layers/library/python/python:3.12.7-alpine3.20
```

9. Automatic Container Startup on Server Restart

Earlier, after server maintenance and a restart, manual intervention was required to restart the Docker containers, resulting in services remaining down until the containers were manually started. This was a manual intervention because it required extra steps to bring the services back up. However, the introduction of a **restart feature** has resolved this problem. The script automatically restarts the Docker containers after the server restarts, ensuring that services are up and running without manual intervention. This automation helps reduce downtime and ensures that the system is fully operational as soon as the server is back online.

```
version: "3.6"

services:

  acp-client-api:
    image: [redacted].dkr.ecr.us-east-1.amazonaws.com/acp-client-api:${CLIENT_IMAGE}"
    tty: true
    volumes:
      - acp-log-volume:/var/log/ai4c
    container_name: acp-client-api
    deploy:
      resources:
        limits:
          memory: 8G
    cpu_shares: 4096
    ports:
      - "9000:9000"
    networks:
      - acp_network
    restart: unless-stopped
```

Add this in docker compose to automate the issue

10.Ensuring Code Quality and Security in CI/CD Pipeline

In the CI/CD pipeline, both **SonarQube** and **Prisma Cloud** scans are integrated to ensure that code quality and security are thoroughly checked before deployment. **SonarQube** analyzes the source code for potential issues, such as bugs, security vulnerabilities, and code smells, helping maintain high code quality and security standards. **Prisma Cloud** then scans the Docker image for any security vulnerabilities, misconfigurations, or compliance issues, ensuring the container is secure and free from known risks. The pipeline is only allowed to proceed with deployment if both scans pass without any critical issues, guaranteeing that only secure and high-quality code is deployed to production. Other security tools can also be integrated in the pipeline.

```
    stage('SonarQube Analysis') {
    if (env.QG_CHECK == "true") {
        def scannerHome = tool 'sonar-scanner'
        withSonarQubeEnv('SonarQube10') {
            sh '''
            export SONAR_TOKEN=${SONAR_TOKEN}
            cd acp-client-api
            $MVN_HOME/bin/mvn sonar:sonar -f pom.xml -X \
              -Dsonar.host.url=http://<server ip>:8091 \
              -Dsonar.projectKey=acp-client-api \
              -Dsonar.projectName=acp-client-api \
              -Dsonar.java.binaries=target \
              -Dsonar.flow.file.suffixes=xml \
              -Dsonar.lang.patterns.xml=flowxml \
              -Dsonar.qualitygate.wait=true \
              -Dsonar.qualitygate.timeout=30
            '''
        }
    }
}

stage('Quality Gate Check') {
    if (env.QG_CHECK == "true") {
        timeout(time: 1, unit: 'HOURS') {
            def qg = waitForQualityGate()
            if (qg.status != 'OK') {
                error "Pipeline aborted due to quality gate failure: ${qg.status}"
            } else {
                echo "Quality Gate passed: ${qg.status}"
            }
        }
    }
}
```

11. Avoid Hardcoding Secrets in Docker Images

Hardcoding secrets in Docker images introduces critical security vulnerabilities. If images with embedded credentials are pushed to public or poorly secured registries, sensitive data can be exposed. Hardcoded secrets can also end up in version control systems, leaving a permanent record even after deletion. Anyone with access to the image can easily extract these secrets using basic commands. Moreover, rotating hardcoded credentials becomes complex, requiring image rebuilds and redeployments, leading to delays and increased risks. This practice also violates compliance standards like PCI DSS, HIPAA, and GDPR, which require secure handling of sensitive data.

AWS Secrets Manager, AWS SSM Parameter Store, HashiCorp Vault, and Jenkins Credentials provide different approaches to secure secret management. AWS Secrets Manager is designed for sensitive data, offering automatic rotation, encryption, and audit logging, making it ideal for production use. AWS SSM Parameter Store allows encrypted storage of secrets and configurations but lacks native rotation, making it more suitable for non-critical data. HashiCorp Vault delivers robust secret management with dynamic secrets, granular access control, and multi-cloud support, perfect for complex environments. Jenkins Credentials focuses on securely managing secrets within CI/CD pipelines but doesn't offer features like automatic rotation or broader integrations.

DevOps can implement secret manager in buildspec.yml.

```
version: 0.2
env:
  secrets-manager:
    CLIENT_TOKEN: "arn:aws:secretsmanager:us-east-1:[redacted]:secret:prisma-[redacted]:SONAR_CLIENT_API"
```

12. Multi-Host Docker Setups

Using docker overlay networks allow containers on different Docker hosts to communicate seamlessly without exposing ports to the public. This is highly beneficial in setups in **containerization for** multi-host Docker environments.

One of the key advantages is **enhanced security**. Overlay networks encapsulate container traffic using **VXLAN (Virtual Extensible LAN)**, isolating internal communication from the external network. Additionally, encryption can be enabled to secure data transmitted between containers.

Overlay networks also simplify **service discovery** by using internal DNS, allowing containers to communicate using service names rather than IP addresses.

They also improve **high availability** and **scalability** by enabling containers to run across multiple hosts while remaining connected on the same network, ensuring resilience and flexibility as services scale.

It can be integrated in **Docker Compose**.

```
    networks:
      - acp_overnet
    restart: unless-stopped

networks:
  acp_overnet:
    name: acp_overnet
    external: true
```

Conclusion

In today's fast-evolving market, adopting Docker security best practices is crucial for developers and DevOps teams in streamlining software delivery. By integrating advanced security measures such as secure secret management, controlled port exposure, and encrypted container communication, development workflows become more resilient and efficient. Developers can focus on building features without constant security concerns, while DevOps teams can enhance deployments and maintain strong governance over infrastructure. These practices not only improve system reliability and compliance but also enable organizations to navigate the demands of modern digital environments with confidence and agility.

References:

Books:

[1] B. Potter and S. Ward, *Docker Security: Virtualization and Container Security*, O'Reilly Media, 2015.
[2] P. Raj and V. Singh, *Learning Docker*, Packt Publishing, 2015.
[3] K. Matthias and S. P. Kane, *Docker: Up & Running*, O'Reilly Media, 2015.
[4] N. Poulton, *Docker Deep Dive*, Independently Published, 2017.

Journal Papers:

[5] G. Combe, R. State, and M. Festor, "To Docker or Not to Docker: A Security Perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54-62, 2016.

[6] Y. Wu, Y. Ding, and Y. Fu, "Vulnerability Analysis and Security Research of Docker Container," *IEEE Int. Conf. on Computational Science and Engineering (CSE)*, pp. 646-653, 2020.

[7] M. You, J. Kim, and S. Shin, "Revisiting Security Landscape of Docker Hub Container Images," *Journal of Korean Institute of Communications and Information Sciences*, vol. 47, no. 3, pp. 321-330, 2022.

[8] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets Revealed in Container Images: An Internet-wide Study on Occurrence and Impact," *arXiv preprint arXiv:2307.03958*, 2023.

[9] S. P. Mullinix, E. Konomi, R. D. Townsend, and R. M. Parizi, "On Security Measures for Containerized Applications Imaged with Docker," *arXiv preprint arXiv:2008.04814*, 2020.