**Hardening Container Images: Best Practices and Examples for Docker**

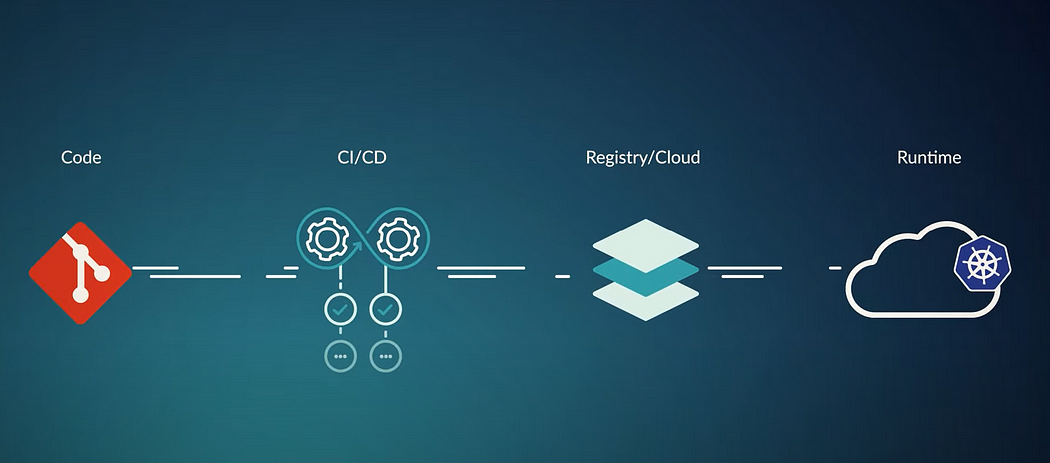
**Introduction**

In the era of cloud computing and microservices, containerization has become a crucial aspect of software deployment. Docker, being one of the most popular containerization platforms, requires careful attention to security.

Container image hardening is a process of securing a container by reducing its attack surface and making it less vulnerable to exploits.

This article will guide you through the best practices that will help a developper for hardening Docker container images.

The security of a Docker images represents just the beginning of a broader picture to secure, as outlined here :



(Code : Application code, Dockerfile code, docker images)

**The best practices discussed in this article are applicable to developers for the purpose of verifying the security of their local Docker images (Code : Application code, Dockerfile code, Docker images).**However, it is crucial for DevOps teams to automatically verify these images using the same technics inside the CI/CD pipelines before any developer pushes a Docker image to a production container registry. This part will be the object of an other article (more details at the end of this article!).

**1. Multi-Stage Builds**

Best Practice: Use multi-stage builds to separate the build environment from the runtime environment. This helps in including only the necessary artifacts in the final image.

Example of Dockerfile:

# Build stage  
FROM node:14 AS build  
WORKDIR /app  
COPY . .  
RUN npm install  
  
# Runtime stage  
FROM alpine:3.14  
COPY --from=build /app /app  
WORKDIR /app  
CMD ["./app"]

This Dockerfile is an example of a multi-stage build. The first stage builds the application in a Node.js environment, and the second stage creates a smaller, lightweight runtime image containing only the built application and its runtime dependencies. The multi-stage approach is used to reduce the final image size, which is beneficial for storage, distribution, and security.

**Tips :**Use COPY Instead of ADD When Writing Dockerfiles. The COPY instruction copies files from the local host machine to the container file system. The ADD instruction can potentially retrieve files from remote URLs and perform unpacking operations. Since ADD could bring in files remotely, the risk of malicious packages and vulnerabilities from remote URLs is increased :

# Example Dockerfile using ADD  
FROM ubuntu:latest  
  
# Potentially risky: ADD can fetch remote files and unpack archives  
ADD http://example.com/mypackage.tar.gz /usr/local/mypackage/  
  
# Rest of the Dockerfile

**2. Use Minimal Base Images**

Best Practice: Start with a minimal base image like Alpine or a slim version of popular distributions. Smaller images contain fewer components, reducing the potential attack surface.

Example:

FROM alpine:3.14

**Tips :**For maximum stability of your build prefer to use specific tags rather than ‘latest’ to ensure consistency and predictability of your images.

Example:

FROM alpine:3.14

If you don’t specify a specific version or tag in your Dockerfile, it will default to using the latest version of the image. This can potentially break your build if there are compatibility issues between your application and the latest image version, as changes in the base image may not be tested or prepared for your specific use case.

***Warning :****This approach can simplify your Dockerfile and ensure consistency, but it also means that you won’t automatically get the latest security patches when you build your image.*

**3. Keep Your Images Up-to-date**

Best Practice: Regularly update your images to include the latest security patches. This can be automated using CI/CD pipelines.

**If you used the “lastest” tag for images in your DockerFile:**

Example: In your CI/CD script:

docker build --pull -t myimage .

This command builds a Docker image using the Dockerfile in the current directory, tags the image as myimage, and ensures that the latest base image is pulled before building.

**If you used specific tags for images in your DockerFile:**

Even if you are using a specific tag version for your base image, you should periodically check for updates to that specific version. If the maintainers of the base image release a new version with security patches, you should update your Dockerfile to test the latest version and see if your application is still working as intented.

**4. Add the HEALTHCHECK Instruction to the Container Image**

The HEALTHCHECK instructions directive tells Docker how to determine if the state of the container is normal. Add this instruction to Dockerfiles, and based on the result of the healthcheck (unhealthy), Docker could exit a non-working container and instantiate a new one.

Add HEALTHCHECK to monitor container health :

HEALTHCHECK --interval=30s --timeout=30s --start-period=5s --retries=3 CMD curl -f http://localhost:8080/health || exit

Then if you run the container using Docker you can use this command to make sure that your unhealty container is restarted :

docker run --restart on-failure [other options] [image name]

***Warning :****Please note that this HEALTHCHECK instructions is not interpreted when running inside Kubernetes. Use Kubernetes livenessprob when deploying your container inside Kubernetes.*

**5. Specify Non-Root User**

Best Practice: Run the container as a non-root user. Containers running as root pose a significant security risk (an hacker having remote command in your running container could install softwares, disable security measures installed in the container, and have a bigger attack surface), especially if they interact with the host system.

Define a user in your Dockerfile and switch to this user before running your application.

Example:

FROM node:14  
# Create app directory  
WORKDIR /usr/src/app  
# Install app dependencies  
COPY package\*.json ./  
RUN npm install  
# Bundle app source  
COPY . .  
# Create a non-root user and switch to it  
RUN adduser -D nonrootuser  
USER nonrootuser  
# Expose the port the app runs on  
EXPOSE 8080  
# Start the app  
CMD [ "node", "app.js" ]

**6. Avoid Leaking Sensitive Information**

Best Practice: Never hardcode sensitive information like passwords or API keys (secrets) in the code of your application or in your Dockerfile.

**When your application needs to retrieve these secrets:**

The recommended approach is to set environment variables when you run a container using docker :

docker run -e MY\_VARIABLE=my\_value my-image

Or like that using kubernetes deployment yaml file (here the environment variable is “DATABASE\_URL”) :

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: nginx-deployment  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: nginx  
 template:  
 metadata:  
 labels:  
 app: nginx  
 spec:  
 containers:  
 - name: nginx  
 image: nginx:latest  
 env:  
 - name: DATABASE\_URL  
 value: "fake-database-url"

Then the code of your application running inside the container will be able to retrive these environment variable.

***For Azure users : i****f you run your docker inside Azure, you can use the managed identity of the service hosting the container to connect to an Azure keyvault to securely retrieve your secrets (*[*example here*](https://learn.microsoft.com/en-us/azure/container-instances/container-instances-managed-identity)*).*

**When you need to access these secrets when building the docker image using your Dockerfile:**

If you really need to use a secret inside a Dockerfile because you need the secret for the build of the docker image (for exemple to connect to a remote private repository) then you can create a file containing the secret and then use it during the build process without leaving traces in the final image.

Example:

FROM alpine   
RUN --mount=type=secret,id=mysecret ./build-script.sh

then Build the image with the secret file “myscret.txt” :

DOCKER\_BUILDKIT=1   
docker build --secret id=mysecret,src=mysecret.txt

The build-script.sh will be able to find the secret as a file in path /run/secrets/mysecret.

***warning :****Inside the “build” Dockerfile never copy a secret inside a file and then delete it. Never pass password in the using –build-arg. Hackers retrieving your docker image will be able to steal these passwords.*

*More info about these risk here :*[*https://pythonspeed.com/articles/docker-build-secrets/*](https://pythonspeed.com/articles/docker-build-secrets/)

*and here :*[*https://pythonspeed.com/articles/docker-secret-scanner/*](https://pythonspeed.com/articles/docker-secret-scanner/)

**7. Utilize .dockerignore File**

Best Practice: Leverage the .dockerignore file to exclude unnecessary files and folders from your Docker build context. This not only speeds up the build process but also prevents potentially sensitive files from being included in the Docker image, reducing the risk of accidental exposure.

Example: Create a .dockerignore file in the root of your project and include entries for files and directories that should not be copied into the image. Common examples include:

.git  
.gitignore  
Dockerfile  
.dockerignore  
README.md  
node\_modules  
npm-debug.log

In this example, version control directories, logs, and local configuration files are excluded. This ensures that only the necessary files are included in the build context, which is particularly important for security-sensitive applications.

Incorporating a well-defined .dockerignore file into your Docker workflow (in the same directory as your Dockerfile) is a simple yet effective step towards more secure and efficient Docker builds.

**8. Compromised Images**

Best Practice: Images can be compromised during development or through insecure registries, allowing attackers to inject malicious code. Use digital image signing and ensure that images are pulled from trusted, secure registries. Implement robust CI security measures to prevent tampering.

Code Example: Enabling Docker Content Trust :

# Enable Docker Content Trust  
export DOCKER\_CONTENT\_TRUST=1

# Pull an image with Docker Content Trust enabled  
docker pull [TRUSTED\_IMAGE\_NAME]

Enabling Docker Content Trust ensures only signed images from trusted publishers are used.

If you use a base image, additional layers, or other images within your Dockerfile, those images should be signed (i.e., they must have a valid signature). Docker will verify the signatures of these images when you build your Docker image.

Tips :Viewing Trust Information:

To view detailed trust information about an image, you can use the Docker Notary tool, which is the underlying technology for Docker Content Trust.

Install the [Notary client](https://github.com/notaryproject/notary/releases)if it’s not already installed.

Use the Notary client to list the signers of a specific image:

notary -s https://notary.docker.io -d ~/.docker/trust list docker.io/[IMAGE\_NAME]

This command will display the signers of the image along with the keys and signatures. You can then verify this against your list of trusted entities or signers.

***For Azure users :****More info info about content trust in Azure (retrieve and push signed docker images from Azure container registries) :*[*https://learn.microsoft.com/en-us/azure/container-registry/container-registry-content-trust*](https://learn.microsoft.com/en-us/azure/container-registry/container-registry-content-trust)

**9. Minimize Exposed Ports**

Best Practice: Only expose the ports that are absolutely necessary for your Docker container. Exposing unnecessary ports increases the attack surface for potential security vulnerabilities. For instance, if a service running on an exposed port has a security flaw, it can be exploited by attackers.

To minimize this risk, you should identify the ports that your application truly needs to function and only expose those. This practice not only enhances security but also simplifies network management and container configuration.

Example:

FROM node:14  
# Create app directory  
WORKDIR /usr/src/app  
# Install app dependencies  
COPY package\*.json ./  
RUN npm install  
# Bundle app source  
COPY . .  
# Create a non-root user and switch to it  
RUN adduser -D nonrootuser  
USER nonrootuser  
# Expose only the necessary port  
# Suppose your application only needs port 8080  
EXPOSE 8080  
# Start the app  
CMD [ "node", "app.js" ]

In this example, EXPOSE 8080 is the key line. It indicates that only port 8080, which is essential for the application, is exposed. All other ports remain unexposed, reducing the likelihood of unauthorized access or attacks on unused or insecure ports.

**10. Dockerfile Misconfigurations**

Best Practice: Images can be misconfigured, leading to security weaknesses, such as inappropriate user permissions or exposed sensitive data. Tools like Hadolint can help analyze Dockerfiles for potential issues.

Code Example: Analyzing Dockerfiles with Hadolint

# Install Hadolint  
apt-get install hadolint  
# Analyze a Dockerfile for best practices  
hadolint /path/to/Dockerfile

This checks the Dockerfile against best practices, helping identify possible misconfigurations.

This is a a good video tutorial to learn how to use hadolint on you Dockerfile on your laptop or in a build pipeline :

Hadolint can be implemented inside an Azure devops CI/CD pipeline :

**[Applying Dockerfile best practices with Hadolint](https://www.kristhecodingunicorn.com/post/dockerfile-linting-with-hadolint/?source=post_page-----e941263cab13--------------------------------" \t "_blank)**

[In this blog post we'll look into how we can ensure that Dockerfiles we create are following best practices with…](https://www.kristhecodingunicorn.com/post/dockerfile-linting-with-hadolint/?source=post_page-----e941263cab13--------------------------------" \t "_blank)

[www.kristhecodingunicorn.com](https://www.kristhecodingunicorn.com/post/dockerfile-linting-with-hadolint/?source=post_page-----e941263cab13--------------------------------" \t "_blank)

**11. Scan you application code for Vulnerabilities and backdoors**

Best Practice: Regularly scan your application code with a scanner that does static code analysis.

Example: Scanning Images with SonarQube

# Navigate to your project directory  
cd /path/to/your/project  
  
# Run SonarScanner analysis  
sonar-scanner

This scans the code of your application for vulnerabilities, aiding in proactive risk mitigation.

Here a good video that shows how to install SonarQube on a local server :

and here a video that shows what type of vulnerabilities SonarQube can detect :

and if you are more of a reader : <https://medium.com/@can.seker/sonarqube-step-by-step-static-code-analysis-implementation-54eac845c486>

SonarQube can be implemented inside an Azure devops CI/CD pipeline :

**12. Scan you Docker image for Vulnerabilities**

Best Practice: Regularly scan your container images for vulnerabilities using tools like Trivy, Clair, or Docker’s own scanning feature.

Example: Scanning Images with Trivy

# Install Trivy  
apt-get install trivy  
# Scan a container image for vulnerabilities  
trivy image [YOUR\_IMAGE\_NAME]

This scans the specified container image for known vulnerabilities, aiding in proactive risk mitigation.

Trivy can be used to scan images, folders, etc. This is a good tutorial to start using Trivy on your laptop :

And this video show how it can be used in a build pipeline :

And if you are more a of a reader :

**[Find Vulnerabilities with Trivy and Publish Them in Azure DevOps](https://medium.com/adessoturkey/find-vulnerabilities-with-trivy-and-publish-them-in-azure-devops-44a9f60cee9e?source=post_page-----e941263cab13--------------------------------" \t "_blank)**

[What is Trivy?](https://medium.com/adessoturkey/find-vulnerabilities-with-trivy-and-publish-them-in-azure-devops-44a9f60cee9e?source=post_page-----e941263cab13--------------------------------" \t "_blank)

[medium.com](https://medium.com/adessoturkey/find-vulnerabilities-with-trivy-and-publish-them-in-azure-devops-44a9f60cee9e?source=post_page-----e941263cab13--------------------------------" \t "_blank)

**[Container Security Scanning with Trivy and Azure DevOps](https://lgulliver.github.io/container-security-scanning-with-trivy-in-azure-devops/?source=post_page-----e941263cab13--------------------------------" \t "_blank)**

[Recently I've been taking a deeper look into how we can bake security scanning and practices into CI/CD pipelines…](https://lgulliver.github.io/container-security-scanning-with-trivy-in-azure-devops/?source=post_page-----e941263cab13--------------------------------" \t "_blank)

[lgulliver.github.io](https://lgulliver.github.io/container-security-scanning-with-trivy-in-azure-devops/?source=post_page-----e941263cab13--------------------------------" \t "_blank)

**13. Continuous Approach**

A fundamental approach to securing container images is to automate the building and testing process.

In short, development teams need a structured and reliable process for building and testing the container images they create. Here’s how this process might look:

1. Developers commit code changes to source control.
2. The CI platform scans the application code and Dockerfile using Hadolint and SonarQube
3. The CI platform halts the container build if it detects vulnerabilities.
4. The CI platform builds the container image.
5. The CI platform pushes the container image to a staging registry.
6. The CI platform invokes a tool like Trivy to scan the image and then a task can execute your container image to detect any unusual behaviors ***(such as unwanted outbound network connections using a***[***technics likes this one***](https://husobee.github.io/http/tcpdump/docker/2016/11/30/troubleshooting-http-docker-tcpdump.html)***or by analyzing the outbound network traffic using a solution like suricata. I have yet to find how to build a live docker image analyzer like an EDR…I know commercial products doing that like Palo Alto Prisma but the challenge would be to build one for “free” :***[***https://www.paloaltonetworks.com/blog/prisma-cloud/image-analysis-sandbox/***](https://www.paloaltonetworks.com/blog/prisma-cloud/image-analysis-sandbox/)***. If you have any idea how to create a tool like that do not hesitate to comment :) ).***
7. The CI platform rejects the container if it finds vulnerabilities or unexpected malicious behaviors
8. If the image passes the policy evaluation and all other tests defined in the pipeline, the image is signed and then pushed to a production registry.

Good video summing up all of that:

**I will soon write an article explaining how to automatize all these steps inside Azure devops so stay tuned by following me :)**

**Conclusion**

Container image hardening is an ongoing process that involves keeping up with best practices, monitoring for new vulnerabilities, and continuously improving the security of your containers. By following these guidelines, you can significantly reduce the risk associated with running Docker containers in production environments.