

# Best practices for securing containerized applications

Andrea Ceccanti  
INFN CNAF

Corso CCR "Docker e Orchestrazione di container"  
June, 17th 2021



# Docker images best practices

# Containers can be secure

But care and attention is required!

A google search for “Docker container security best practices” yields several results. I will talk a bit about some of the key advice I found in the following resources:

[https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)

<https://docs.docker.com/develop/dev-best-practices/>

<https://docs.docker.com/engine/security/>

<https://sysdig.com/blog/dockerfile-best-practices/>

# Use trusted base images

Always check the images you extend. Extend trusted, certified base images.

The screenshot displays the Docker Hub website interface. At the top, there is a search bar with the text "search for great content (e.g., mysql)" and navigation links for "Explore", "Pricing", "Sign In", and "Sign Up". Below the search bar, there are tabs for "Docker", "Containers", and "Plugins".

The main content area shows search results for "ubuntu" and "alpine" images. The "ubuntu" image is highlighted as an "OFFICIAL IMAGE" with over 10 million downloads and 10,000+ stars. It is described as a Debian-based Linux operating system. The "alpine" image is also an "OFFICIAL IMAGE" with over 10 million downloads and 7.6K stars, described as a minimal Docker image based on Alpine Linux.

On the left side, there are filters for "Verified Publisher" and "Official Images" (both checked). Below these are categories such as "Analytics", "Application Frameworks", "Application Infrastructure", "Application Services", "Base Images", "Databases", "DevOps Tools", "Featured Images", and "Messaging Services".

At the bottom center, the number "4" is displayed.

# Update your images frequently

Use base images that are frequently updated, and rebuild yours on top of them.

As new security vulnerabilities are discovered continuously, it is a general security best practice to stick to the latest security patches.

# Reduce attack surface: keep images minimal

Use multi-stage builds and leverage minimal base images. If possible, use distroless base images

```
FROM rust:1.41.0 as build-env
WORKDIR /app
ADD . /app
RUN cargo build --release
```

```
FROM gcr.io/distroless/cc
COPY --from=build-env /app/target/release/hello-world-distroless /
CMD ["/hello-world-distroless"]
```

# Do not run your application as root

Do not run your application as root within a container

- always use the USER instruction in your Dockerfile
- Provide appropriate file system permissions in the locations where the process will be reading or writing

```
FROM alpine:3.12
# Create user and set ownership and permissions as required
RUN adduser -D myuser && chown -R myuser /myapp-data
# ... copy application files
USER myuser
ENTRYPOINT ["/myapp"]
```

# Do not bind the user to a specific UID

Some platforms (e.g., Openshift) will use a random UID when running containers.

Use the tmp dir to write temporary data and make resources world readable

```
...  
USER myuser  
ENV APP_TMP_DATA=/tmp  
ENTRYPOINT ["/myapp"]
```



# Make executables root owned and non-writable

This will block the executing user from modifying existing binaries or scripts, which could enable different attacks.

```
...  
WORKDIR $APP_HOME  
COPY --chown=app:app app-files/ /app  
USER app  
ENTRYPOINT /app/my-app-entrypoint.sh
```

# Make executables root owned and non-writable

This will block the executing user from modifying existing binaries or scripts, which could enable different attacks.

```
...
```

```
WORKDIR $APP_HOME
```

```
COPY --chown=app:app app-files/ /app
```

```
USER app
```

```
ENTRYPOINT /app/my-app-entrypoint.sh
```



# Prevent confidential data leaks

Never put any secret or credentials in the Dockerfile instructions.

Be extra careful with files that get copied into the container. Even if a file is removed in a later instruction in the Dockerfile, it can still be accessed on the previous layers as it is not really removed, only “hidden” in the final filesystem.

Don't include confidential information or configuration values that tie them to some specific environment (i.e., production, staging, etc.).

# Favour COPY over ADD

Both the ADD and COPY instructions provide similar functions in a Dockerfile. However, COPY is more explicit.

Use COPY unless you really need the ADD functionality, like to add files from an URL or from a tar file. COPY is more predictable and less error prone.

# Understand the docker build context

Only include the minimal and necessary information in the docker build context.  
Use the .dockerignore file, and use a dedicated folder for Docker image assets

```
docker build -t myimage .
```



```
docker build -t myimage assets/
```



# Do not install unnecessary packages

To reduce complexity, dependencies, file sizes, and build times, avoid installing extra or unnecessary packages just because they might be “nice to have.” For example, you don’t need to include a text editor in a database image.

# Decouple applications

Each container should have only one concern.

Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers.

For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.

# Minimize the number of layers

In recent Docker version, only the instructions RUN, COPY, ADD create layers. Other instructions create temporary intermediate images, and do not increase the size of the build.

Where possible, **use multi-stage builds**, and only copy the artifacts you need into the final image.



# Continuously build your images

When you check in a change to source control or create a pull request, use a CI/CD pipeline to automatically build and tag a Docker image and test it.

# Properly tag your images

Follow a coherent and consistent tagging policy. Document your tagging policy so that image users can easily understand it.

Container images are a way of packaging and releasing a piece of software. Tagging the image lets users identify a specific version of your software in order to download it. For this reason, tightly link the tagging system on container images to the release policy of your software

Examples:

- Include a version number following semantic version in your tags
- Use the git commit SHA hash as a tag for your code

# Use static image tags in production

Avoid “moving” tags like `latest`, the application could change without you being aware of it and break your system, i.e. the main benefit of immutability of the infrastructure for which we use containers are lost!

There’s also an image caching and scalability aspect: using fixed tags reduces network traffic and can avoid hitting DockerHub download limits.

# Scan images for vulnerabilities

DockerHub provides this service for Docker trusted images.

You can have a similar functionality on a local Harbor registry