Best practices for securing containerized applications

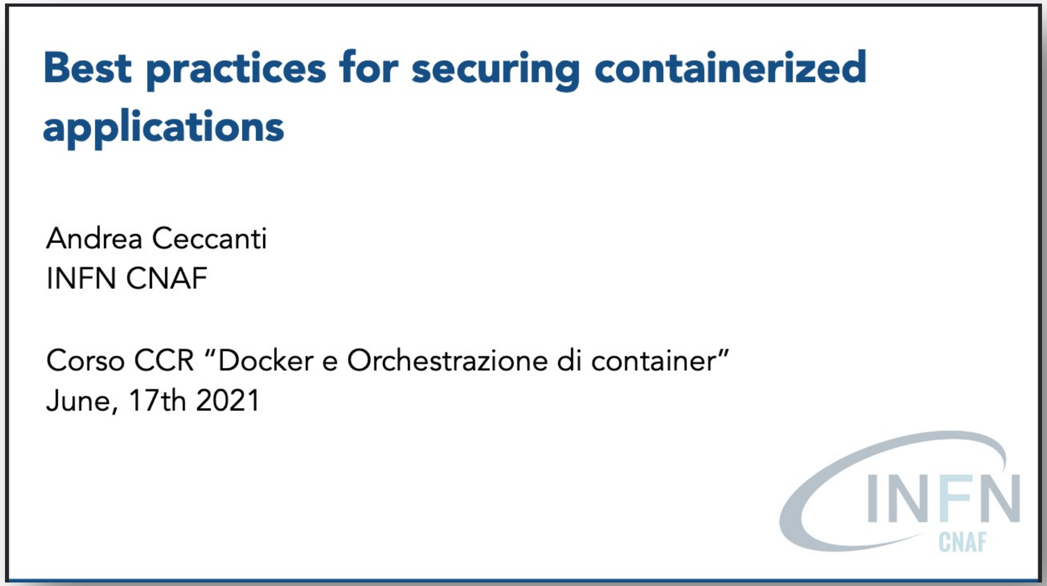
Andrea Ceccanti, Doina Cristina Duma

CC-BY 4.0

Before we begin (about these slides)

These slides were prepared by **Andrea Ceccanti**

Kudos to the author!



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/

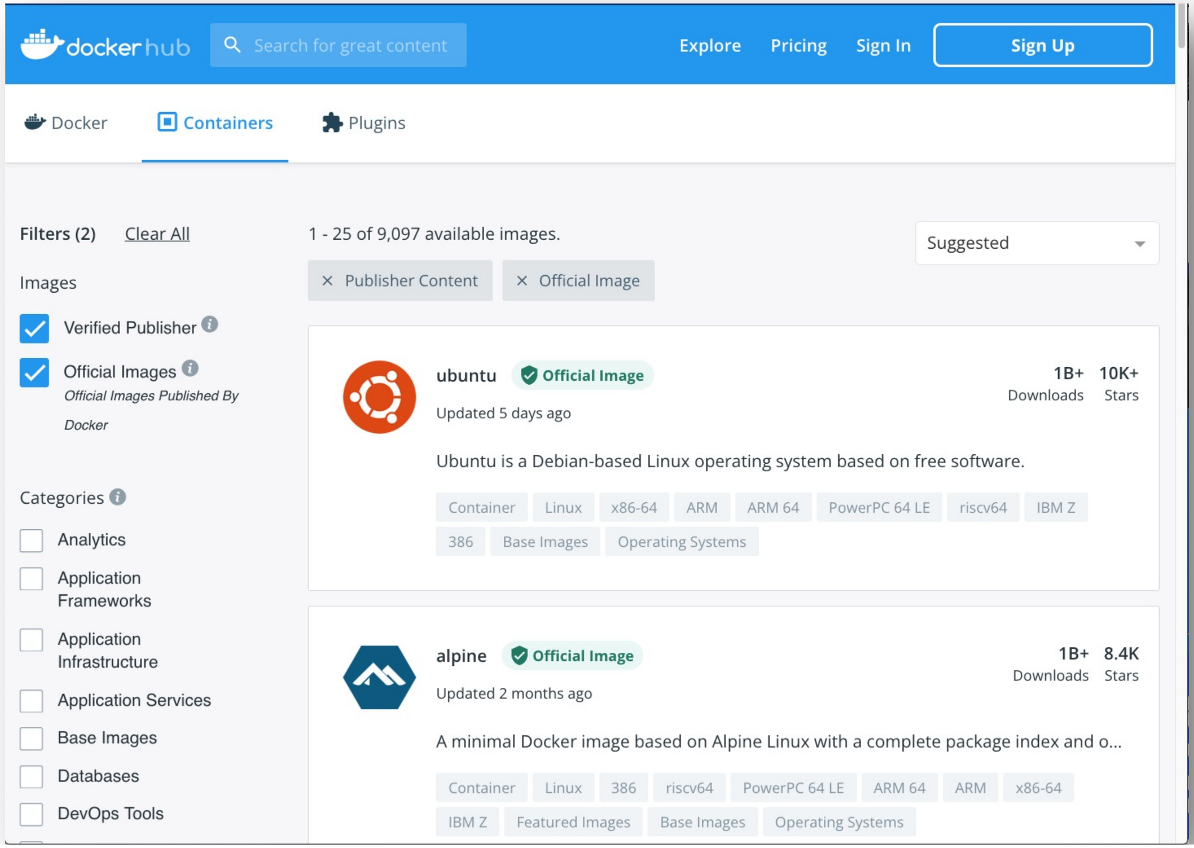
https://docs.docker.com/develop/dev-best-practices/

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.



Filters (2) Clear All

Images

- ✓ Verified Publisher
- ✓ Official Images
  Official Images Published By Docker

Categories

- Analytics
- Application Frameworks
- Application Infrastructure
- Application Services
- Base Images
- Databases
- DevOps Tools

1 - 25 of 9,097 available images.

Suggested

× Publisher Content × Official Image

ubuntu Official Image — 1B+ Downloads, 10K+ Stars

Updated 5 days ago

Ubuntu is a Debian-based Linux operating system based on free software.

Container | Linux | x86-64 | ARM | ARM 64 | PowerPC 64 LE | riscv64 | IBM Z | 386 | Base Images | Operating Systems

alpine Official Image — 1B+ Downloads, 8.4K Stars

Updated 2 months ago

A minimal Docker image based on Alpine Linux with a complete package index and o...

Container | Linux | 386 | riscv64 | PowerPC 64 LE | ARM 64 | ARM | x86-64 | IBM Z | Featured Images | Base Images | Operating Systems

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"]
```

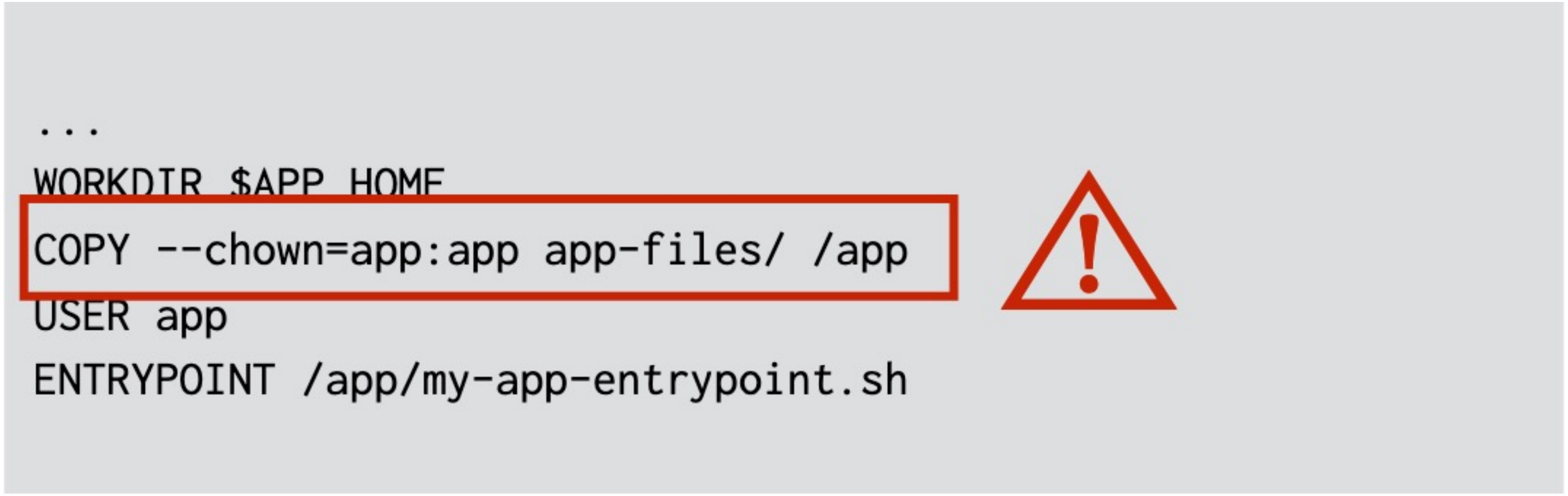
Make executables root owned and non-writable

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

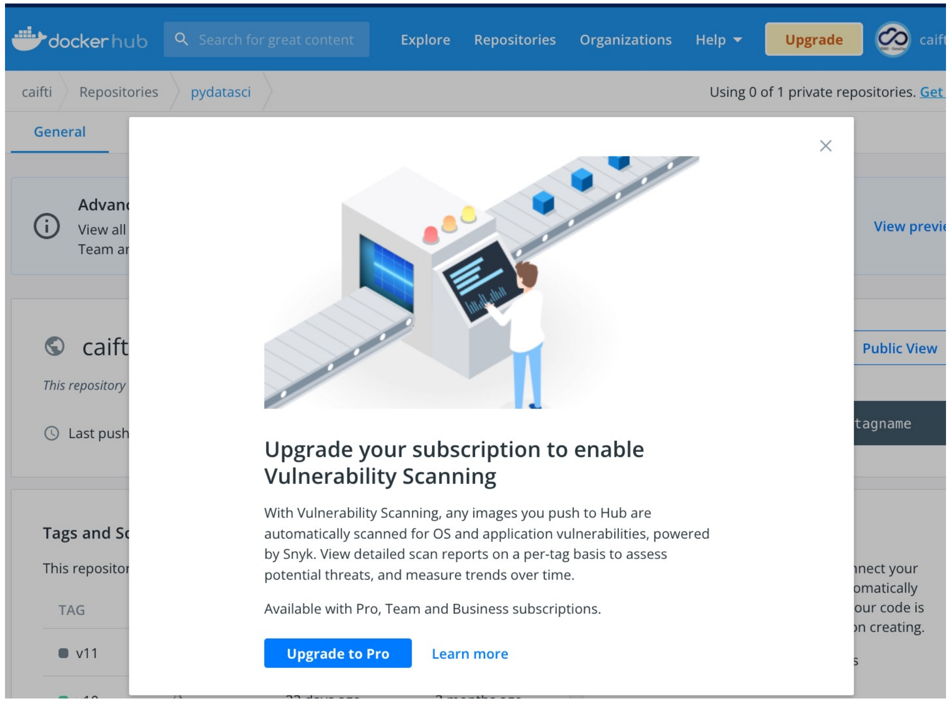
Use static image tags in production

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



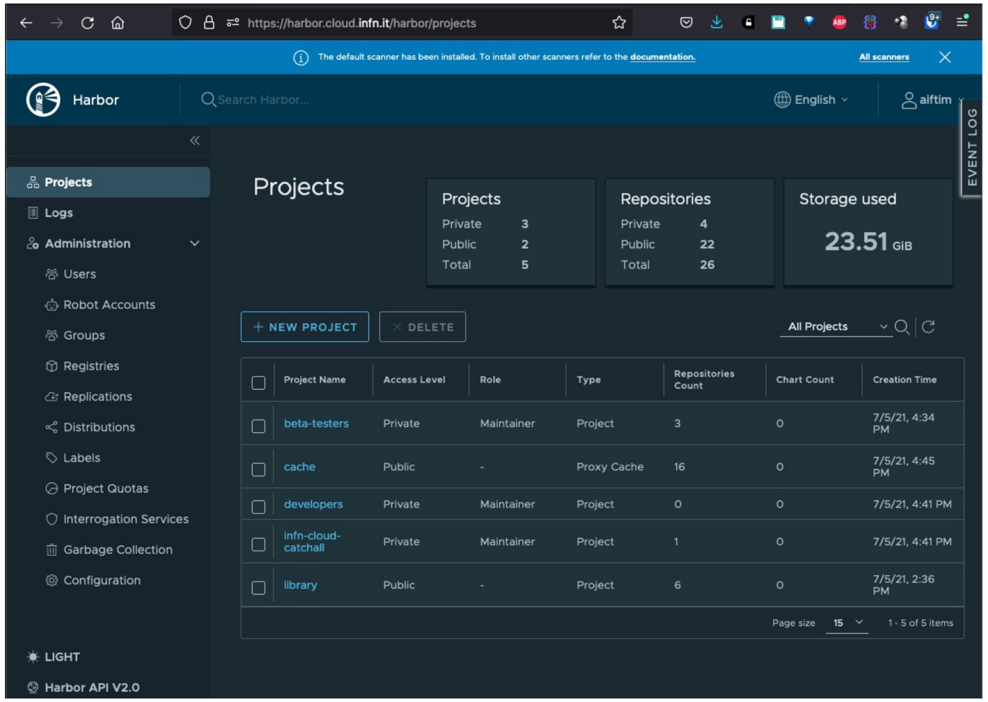
Using 0 of 1 private repositories. Get

Upgrade your subscription to enable Vulnerability Scanning

With Vulnerability Scanning, any images you push to Hub are automatically scanned for OS and application vulnerabilities, powered by Snyk. View detailed scan reports on a per-tag basis to assess potential threats, and measure trends over time.

Available with Pro, Team and Business subscriptions.

Upgrade to Pro   Learn more



The default scanner has been installed. To install other scanners refer to the documentation.

Projects

Projects	
Private	3
Public	2
Total	5

Repositories	
Private	4
Public	22
Total	26

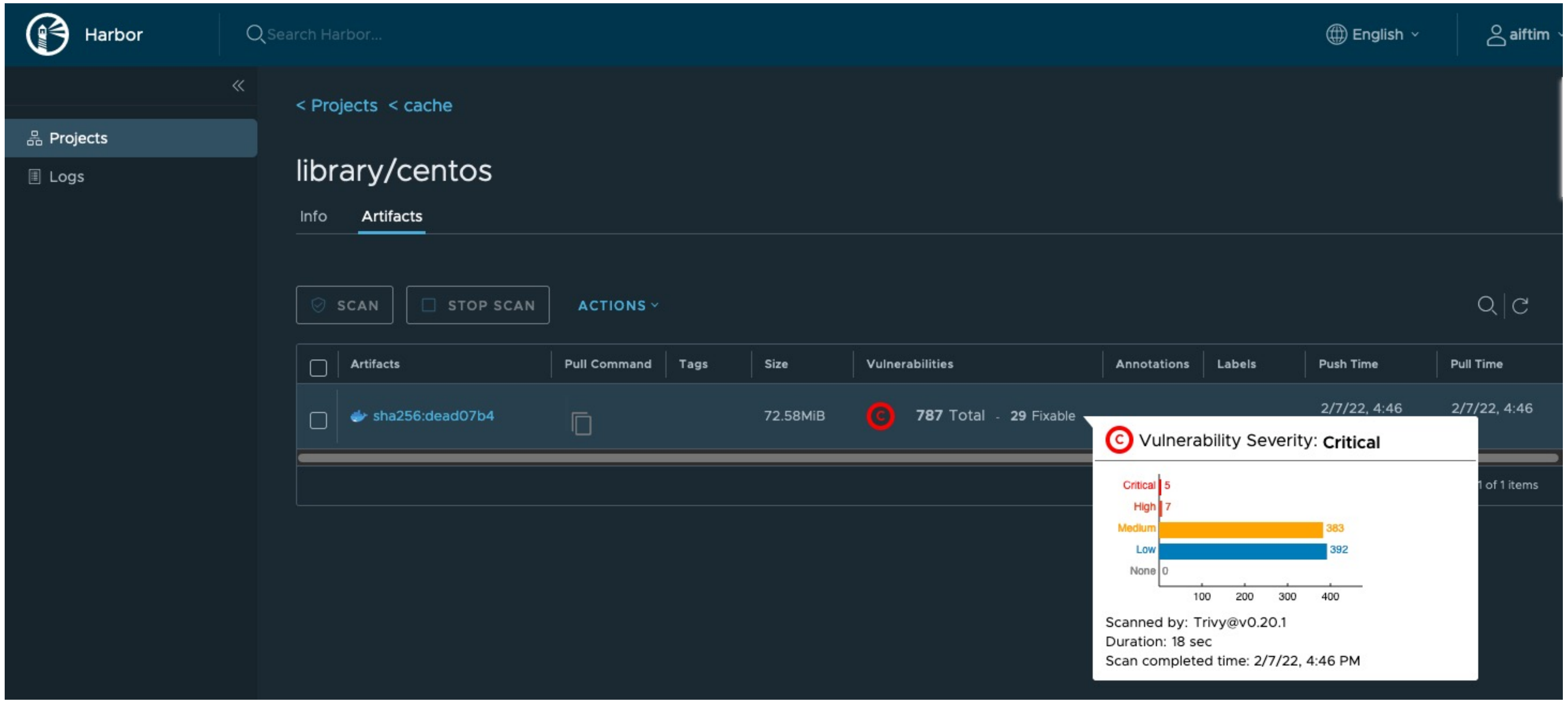
Storage used 23.51 GiB

Project Name	Access Level	Role	Type	Repositories Count	Chart Count	Creation Time
beta-testers	Private	Maintainer	Project	3	0	7/5/21, 4:34 PM
cache	Public	-	Proxy Cache	16	0	7/5/21, 4:45 PM
developers	Private	Maintainer	Project	0	0	7/5/21, 4:41 PM
infn-cloud-catchall	Private	Maintainer	Project	1	0	7/5/21, 4:41 PM
library	Public	-	Project	6	0	7/5/21, 2:36 PM

Page size 15   1 - 5 of 5 items

Scan images for vulnerabilities

DockerHub provides this service for Docker trusted images.
You can have a similar functionality on a local Harbor registry



	Artifacts	Pull Command	Tags	Size	Vulnerabilities	Annotations	Labels	Push Time	Pull Time
	sha256:dead07b4			72.58MiB	787 Total - 29 Fixable			2/7/22, 4:46	2/7/22, 4:46

Vulnerability Severity: Critical

Critical	5
High	7
Medium	383
Low	392
None	0

Scanned by: Trivy@v0.20.1
Duration: 18 sec
Scan completed time: 2/7/22, 4:46 PM

“take away”

Avoid unnecessary privileges

- Avoid running containers as root
- Don’t bind to a specific UID
- Make executables owned by root and not writable

Reduce attack surface.

- Leverage multistage builds
- Use distroless images, or build your own from scratch
- Update your images frequently
- Watch out for exposed ports

Prevent confidential data leaks

- Never put secrets or credentials in Dockerfile
- Prefer COPY over ADD
- Be aware of the Docker context, and use .dockerignore

Beyond image building.

- Protect the docker socket and TCP connections
- Sign your images, and verify them on runtime
- Avoid tag mutability
- Don’t run your environment as root
- Include a health check
- Restrict your application capabilities

Others

- Reduce the no. of layers, and order them intelligently
- Add metadata and labels
- Leverage linters to automatize checks
- Scan your images locally during development

Questions?