

Best practices, sicurezza e ottimizzazioni

Gioacchino Vino (INFN Bari)
gioacchino.vino@ba.infn.it

- Securing the Host Operating System
- Securing Docker Images
- Best Practices in Docker Images
- Securing Docker Container Runtime

Why?

- If attackers compromise the host operating system (OS), they may compromise all processes on the OS, including the container runtime.
- You should design the base OS to run the container engine only, with no other processes that could be compromised.

1) Choosing an OS

BEST PRACTICE: container-specific host OS. Typically include by default:

- Security features
- Automated updates
- Image hardening

MOST POPULAR CHOICE: Linux distribution (general-purpose OSs)

- You need to manage every security feature independently
- Have unnecessary system services or non-containerized applications
- Need consistently scan and monitor your host operating system for vulnerabilities
- Updates

2) Do not expose the Docker daemon socket (even to the containers)

The Docker daemon socket is a Unix network socket that facilitates communication with the Docker API.

By default, this socket is owned by the root user.

If anyone else obtains access to the socket, they will have permissions equivalent to root access to the host.

First Rule: Reduce attack surface

It is a Dockerfile best practice to **keep the images minimal**.

To reduce the attack surface:

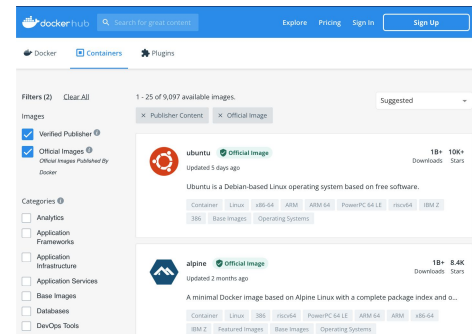
- Avoid including unnecessary packages
 - The more components you include inside a container, the more exposed your system will be and the harder it is to maintain, especially for components not under your control.
- Exposing ports
 - Every opened port in your container is an open door to your system
 - Expose only the ports that your application needs and avoid exposing ports like SSH (22).

1) Use an official and trusted Docker image as a base image

- Carefully choose the base for your images (the FROM instruction): building on top of untrusted or unmaintained images will inherit all of the problems and vulnerabilities from that image into your containers.
- Instead of taking a base operating system image and installing packages you need for your application, use the official node image for your application, if available

Improvements:

- Cleaner Dockerfile
- Official and verified image, which is already built with the best practices



Securing Docker Images

2) Use Small-Sized Official Images

When choosing a Python image, you will see there are actually multiple official images. Not only with different version numbers, but also with **different operating system distributions**.

If you chose an image based on a **full-blown OS distribution**:

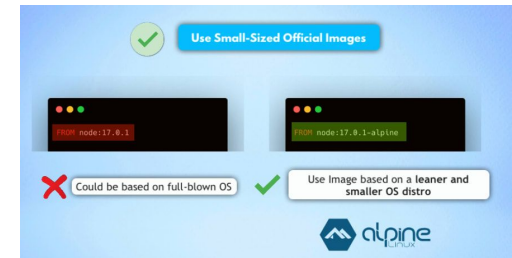
- You will have a bunch of tools already packaged
- The **image size will be larger**, but you **might not need most of these tools** in your application images
- You need to consider the security aspect

```

• 3.11.0rc1-bullseye, 3.11-rc-bullseye
• 3.11.0rc1-slim-bullseye, 3.11-rc-slim-bullseye, 3.11.0rc1-slim, 3.11-rc-slim
• 3.11.0rc1-buster, 3.11-rc-buster
• 3.11.0rc1-slim-buster, 3.11-rc-slim-buster
• 3.11.0rc1-alpine3.16, 3.11-rc-alpine3.16, 3.11.0rc1-alpine, 3.11-rc-alpine
• 3.11.0rc1-alpine3.15, 3.11-rc-alpine3.15
• 3.11.0rc1-windowsservercore-ltsc2022, 3.11-rc-windowsservercore-ltsc2022
• 3.11.0rc1-windowsservercore-1809, 3.11-rc-windowsservercore-1809

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.9.6-alpine3.14	f773016f760e	3 days ago	45.1MB
python	3.9.6-slim	907fc13ca8e7	3 days ago	115MB
python	3.9.6-slim-buster	907fc13ca8e7	3 days ago	115MB
python	3.9.6	cba42c28d9b8	3 days ago	886MB
python	3.9.6-buster	cba42c28d9b8	3 days ago	886MB



2) Use Small-Sized Official Images

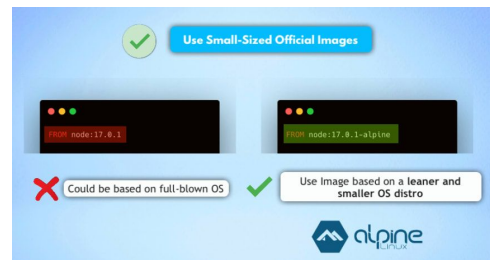
When choosing a Python image, you will see there are actually multiple official images. Not only with different version numbers, but also with **different operating system distributions**.

If a **smaller images** is chosen:

- You need **less storage space** and of course you can transfer the images faster
- You're also minimizing the attack surface and making sure that you build **more secure images**.
- **Distroless** are designed to contain only the minimal set of libraries required to run Go, Python, or other frameworks

- [3.11.0rc1-bullseye](#), [3.11-rc-bullseye](#)
- [3.11.0rc1-slim-bullseye](#), [3.11-rc-slim-bullseye](#), [3.11.0rc1-slim](#), [3.11-rc-slim](#)
- [3.11.0rc1-buster](#), [3.11-rc-buster](#)
- [3.11.0rc1-slim-buster](#), [3.11-rc-slim-buster](#)
- [3.11.0rc1-alpine3.16](#), [3.11-rc-alpine3.16](#), [3.11.0rc1-alpine](#), [3.11-rc-alpine](#)
- [3.11.0rc1-alpine3.15](#), [3.11-rc-alpine3.15](#)
- [3.11.0rc1-windowsservercore-ltsc2022](#), [3.11-rc-windowsservercore-ltsc2022](#)
- [3.11.0rc1-windowsservercore-1809](#), [3.11-rc-windowsservercore-1809](#)

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
python	3.9.6-alpine3.14	f773016f760e	3 days ago	45.1MB
python	3.9.6-slim	907fc13ca8e7	3 days ago	115MB
python	3.9.6-slim-buster	907fc13ca8e7	3 days ago	115MB
python	3.9.6	cba42c28d9b8	3 days ago	886MB
python	3.9.6-buster	cba42c28d9b8	3 days ago	886MB



3) Do Not Install Unnecessary Packages in the Container

Do not install packages outside the scope and purpose of the container:

It will reduce:

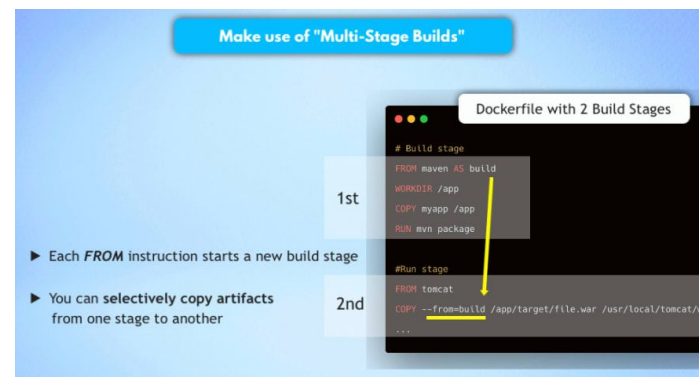
- Container size
- Attack surface
- Complexity
- Dependencies
- File sizes
- Build times

4) Make use of Multi-Stage Builds

- There are some contents in your project that you **need** for building the image but **not** when you running the application.
- Multi-Stage builds **separate** the build stage from the runtime stage
- Only the final stage is used to create the image

Improvements:

- Separation of Build Tools and Dependencies from what's needed for runtime
- Less dependencies and reduced image size



5) Use the Least Privileged User

- By default, Docker runs container processes as root inside of a container.
- This introduces a security issue, because the container will potentially have root access on the Docker host.
- It will make it **easier for an attacker to escalate privileges on the host**
- To prevent this, the best practice is:
 - **create a dedicated user** and a dedicated group in the Docker image
 - **run the application** inside the container **with that user** (USER directive)
- the Docker daemon and the container itself is still running with root privileges

```
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"]
```

6) Make executables owned by root and not writable

- For every executable in a container to be owned by the root user (even if it is executed by a non-root user) and should not be world-writable.
- The user only needs execution permissions on the file, **not ownership**.
- This will block the executing user from modifying existing binaries or scripts
- It's a practice for enforcing container **immutability**.
- You can prevent your running application from being accidentally or maliciously modified.

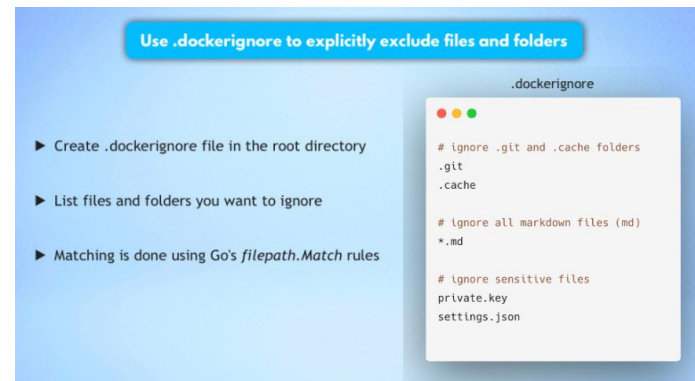
```
WORKDIR $APP_HOME
COPY app-files/ /app
RUN 755 /app/my-app-entrypoint.sh
USER app
ENTRYPOINT /app/my-app-entrypoint.sh
```

7) Use `.dockerignore` file

- Usually when we build the image, we don't need everything we have in the project to run the application inside
- We **don't need the auto-generated folders**, like targets or build folder, we don't need the readme file etc.
- Using a `.dockerignore` file where **list all the files and folders that we want to be ignored** and when building the image

Improvements:

- Reduced image size



The slide features a blue background with a white text box on the left and a code editor window on the right. The text box contains three bullet points: 'Create .dockerignore file in the root directory', 'List files and folders you want to ignore', and 'Matching is done using Go's filepath.Match rules'. The code editor window shows the content of a .dockerignore file with three sections: ignoring .git and .cache folders, ignoring all markdown files, and ignoring sensitive files like private.key and settings.json.

Use `.dockerignore` to explicitly exclude files and folders

- ▶ Create `.dockerignore` file in the root directory
- ▶ List files and folders you want to ignore
- ▶ Matching is done using Go's `filepath.Match` rules

```
.dockerignore
# ignore .git and .cache folders
.git
.cache

# ignore all markdown files (md)
*.md

# ignore sensitive files
private.key
settings.json
```

Best Practices in Docker Images

1) Use specific Docker image versions

Using the latest tag:

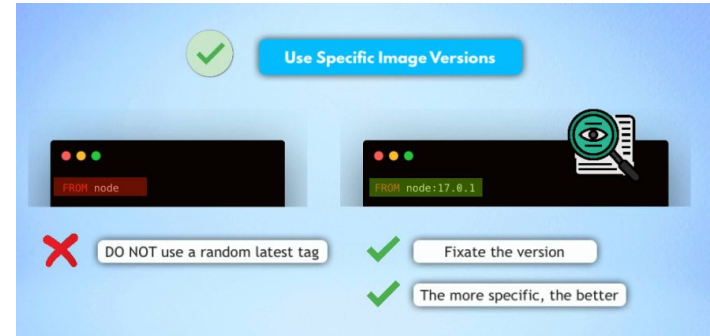
- You might get a different image version as in the previous build
- The new image version may break stuff
- It is unpredictable, causing unexpected behavior

Deploy your own application with a specific version!

And the rule here is: **the more specific the better**

Improvements:

- Transparency to know exactly what version of the base image you're using



Best Practices in Docker Images

2) 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 Docker could exit a non-working container and instantiate a new one.

Best Practices in Docker Images

3) Optimize caching for image layers when building an image

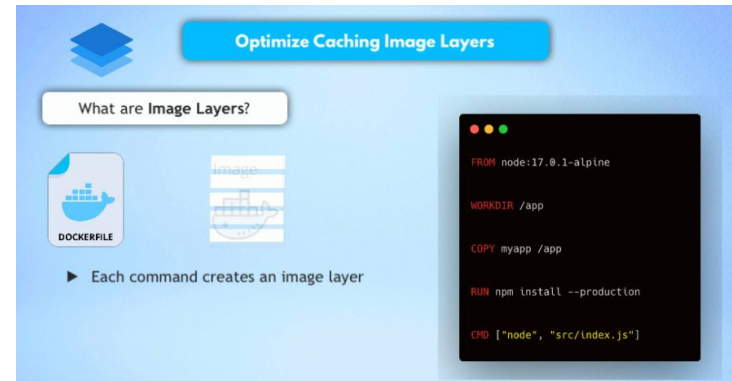
An image is built based on a Dockerfile.

Each command or instruction creates an image layer

- RUN, COPY, and ADD each create layers
- Each layer contains the differences from the previous layer.
- Layers increase the size of the final image.

Best practice:

- Combine related commands.
- Remove unnecessary files in the same RUN step that created them.
- Minimize the number of times apt-get upgrade is run since it upgrades all packages to the latest version.



Best Practices in Docker Images

3) Optimize caching for image layers when building an image

- Each layer will get cached by Docker and when you rebuild your image, if your Dockerfile hasn't changed, Docker will just use the cached layers to build the image.
- Once a layer changes, all following layers have to be re-created as well

Best practice is:

Order your commands in the Dockerfile **from the least to the most frequently changing commands** to take advantage of caching and this way optimize how fast the image gets built.

```
FROM python:3.9-slim
WORKDIR /app
COPY sample.py .
COPY requirements.txt .
RUN pip install -r /requirements.txt
```

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r /requirements.txt
COPY sample.py .
```

Best Practices in Docker Images

4) Avoid Storing Secrets in Images

- Secrets are sensitive pieces of information such as passwords, database credentials, SSH keys, tokens, and TLS certificates, to name a few.
- These **should not** be baked into your images without being encrypted since unauthorized users who gain access to the image can merely examine the layers to extract the secrets
- Never put any secret or credentials in the Dockerfile instructions, especially if you're pushing the images to a public registry
- Instead, they should be injected via:
 - Environment variables (at run-time)
 - Build-time arguments (at build-time)
 - An orchestration tool like Docker Swarm (via Docker secrets) or Kubernetes (via Kubernetes secrets)
 - Encrypted shared volumes

5) Prefer COPY Over ADD

- Both commands allow you to copy files from a specific location into a Docker image:

```
ADD <src> <dest>
```

```
COPY <src> <dest>
```

- COPY is used for copying local files or directories from the Docker host to the image.
- ADD can be used also for:
 - downloading external files.
 - If you use a compressed file as the <src> parameter, ADD will automatically unpack the contents to the given location.
- Use COPY unless you really need the ADD functionality, COPY is more predictable and less error prone

6) Don't bind to a specific UID

- Openshift, by default, will use random UIDs when running containers.
- Forcing a specific UID (i.e., the first standard user with UID 1000) requires adjusting the permissions of any bind mount, like a host folder for data persistence. Alternatively, if you run the container (-u option in docker) with the host UID, it might break the service when trying to read or write from folders within the container.

Best Practices in Docker Images

7) Continuous Approach

- Automate building and testing
- Set up the tooling to analyze images continuously:
 - Git push
 - docker build
 - Scan
 - Production
- Check if:
 - new security vulnerabilities are discovered
 - docker scan
 - other tools
 - your base image has been rebuild (with latest security patches)
- and rebuild your image

8) Use static image tags in production

- 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
- 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

1) Do Not Use Privileged Containers

2) Do Not Expose Unused Ports

3) Do Not Share the Host's Network Namespace

- Use `--net=host` only when strictly needed, not use an isolated network layer

4) Add resource limits

- Avoid container consuming all the memory or CPUs and starve other applications.

5) Set On-Failure Container Restart Policy

- `--restart on-failure[:max-retries]`

6) Set filesystem and volumes to read-only

- `docker run --read-only --tmpfs /tmp alpine sh -c 'echo "whatever" > /tmp/file'`

7) Run Docker in Rootless Mode