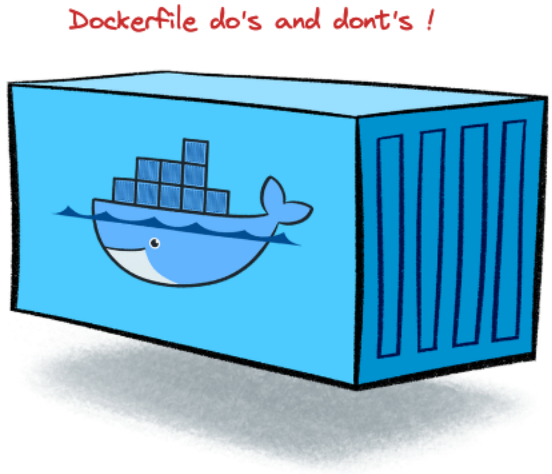
# Container Image Security Best Practices

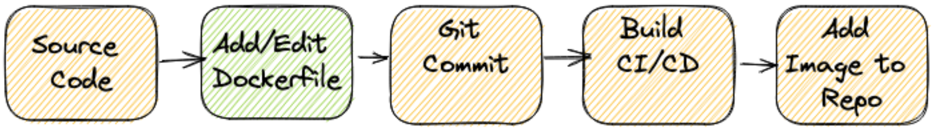
By: SourceBahn Security



## Introduction

Modern application development continues to evolve. Developers not only need to learn how to create APIs, microservices or web applications, they may also need to know how to containerize their work for deployment into Kubernetes clusters or similar. This extra "containerize" (i.e. build a Docker image) step is riddled with gotchas, many of which we will cover in this article.

The developer workflow is typically a process as follows (i.e. includes creating a Docker image definition file called "Dockerfile"):



When developers containerize their application into a Docker image, the first thing they must realize is that they are **no longer deploying their application by itself**. To create a Docker image, developers must bundle their application with additional third-party components. The most important one being the "base OS" image. The base OS image itself may come from some public repository such as https://hub.docker.com. As such, it may be considered "untrusted" unless provided by a trusted vendor. Along with this base OS image, developers may often need

to add additional components as required by the application (i.e. Java, Python, NodeJS, etc.). There are several best practices that aim to address security issues that derive from bundling your application with these additional third-party components. These additional components now become part of the application supply chain (whether you like it or not).
Docker image security concerns exist at two levels: “build-time” as well as “container runtime”. This article describes industry best practices in creating or editing a “Dockerfile” used during the image “**build-time**” process, the contents of which dictate what third-party components are bundled, the overall size and complexity of the resulting image. The important "container runtime” security best practices (i.e. Kubernetes Pod deployments, security context, admission controller, OPA, service mesh) are beyond the scope of this article and will be discussed separately, along with vulnerability management and Shift-Left security best practices for developers.

You may refer to the Docker.com website for additional guidance on writing “Dockerfiles”:
https://docs.docker.com/develop/develop-images/dockerfile_best- practices/

## Best Practice #1: Use Trusted Base Images

The risk of supply chain attacks remains a top security issue for Docker, open-source community and all developers involved. Every Docker image that you build may need to bundle with some base OS image – one must choose wisely!.

Which base OS image you ask?

You have countless options here, enough to make your head spin. Base OS options are offered in many flavors of the Linux kernel (large, slim and anywhere in-between):

- Alpine
- Ubuntu
- CentOS
- Debian

You will find countless distributions that come from public repositories. Beware: some public repositories have been known to contain tainted base OS images containing malware or crypto-mining modules and should not be trusted. Base OS images should first be vetted by experienced security personnel before adoption by your organization.
Yes……it is the wild west out there!

**Best practice**: choose a minimal lightweight base OS image (i.e. -slim version) from a reputable source, then overlay with the required runtime libraries. Some distributions may offer the perfect bundle, with popular tools like Python, NodeJS or JRE already included. These can be very tempting. However, these bundles may not necessarily be maintained regularly, thus exposing your container to a multitude of vulnerabilities derived from some old snapshot in time. There may be little to no trade-off here. Best practice is to have a private repository that offers pre-hardened vetted base OS images which are updated regularly by an experienced middleware and security team. Favor using a base OS image that is small, contains limited or no shell commands at all.

Your Dockerfile may look as follows:

```
FROM containers.mycorp.com/alpine

WORKDIR /app
COPY package.json /app RUN npm install
CMD [“node”, “index.js”]
```

Next, ask yourself the following:

- Do I need a full JDK (with compiler/debug tools) or will the JRE suffice?
- Do I really need CLI tools: openssl, curl, wget, ssh-client?
- Plan to remotely connect via SSH in Production? (i.e. not enforcing immutable images?)

Answering “Yes” to any of the above is absolutely a **Red Flag!**

If any additional components are needed, it is best practice to specify each one separately via the RUN “apk”, “apt-get”, “yum” or similar statement in the “Dockerfile”. By default, the package manager will always download the LATEST version of the requested components, thus preventing outdated versions from being introduced during the image **build-time** process.

The following example shows how to bundle “Nginx”, allowing granular control over the package selection while always fetching the most up-to-date version:

```
FROM containers.mycorp.com/alpine

RUN apk add nginx
COPY index.html /var/www/html/
EXPOSE 80
CMD [“nginx”, “-g”, “daemon off;”]
```

For a **Spring Boot microservice** (REST endpoint), you may consider the following:

```
FROM containers.mycorp.com/alpine

# /dev/urandom is used as entropy source (security best practice)
RUN apk add --no-cache openjdk21-jre \
&& echo "securerandom.source=file:/dev/urandom" >> \
/usr/lib/jvm/default-jvm/jre/lib/security/java.security

EXPOSE 8080
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

## Best Practice #2: Do Not Upgrade the System Packages

Avoid running commands such as:

**RUN apt-get upgrade**

Doing so will only cause traceability and predictability issues. The above command essentially updates all software/base OS dependencies. The base image will no longer be the same. When using a trusted base image, it is best practice to pin this base image and leave it alone.

Restrict the use of the package manager only to:

"apt/apk **update**"
"apt/apk **add**"

## Best Practice #3: Use Fixed Tags For Immutability

Let us first understand how Docker pulls images upon request.
The following FROM commands are functionally equivalent:

FROM **alpine** *(if no tag-->defaults to "latest")*
or
FROM **alpine:latest**
or
FROM **alpine:3.17.1** *(at the time of this writing)*

Tags are used to manage versions of Docker images. In the above example, the “latest” tag tells Docker to pull/fetch the latest version of that image. But “*latest”* is a moving target. As a result, multiple images may end up having a “*latest*” tag, resulting in wide-spread confusion and inconsistent behavior in automated builds. Beware of this consequence, as it will abstract the real underlying version number and make it difficult to trace back or debug when problems occur.

Consider the event of a Kubernetes deployment gone bad and a rollback is needed. The tag ambiguity may cause operational issues and threaten business continuity!

**Best practice is to always use tags - with actual values that are meaningful to anyone in DevSecOps**.

## Best Practice #4: Delete Cached Package Files

This is a hygiene task that aims to optimize the size of your Docker image. When using the RUN command to invoke a package manager (i.e. apk, apt, yum), the process first downloads the remote packages to a local cache directory (i.e. /var/cache/*). After the installation has completed, this cache directory usually stays behind consuming unnecessary disk space.

You may prevent this by using the option **“—no-cache”** when invoking the package manager as follows:

```
FROM containers.mycorp.com/alpine:tagName
RUN apk add --no-cache <package-name>
```

The “—no-cache” argument/parameter is functionally equivalent to:

“RUN rm /var/cache/apk/*”

Alternately, you may handle the cleanup step yourself as follows:

```
FROM containers.mycorp.com/alpine:tagName
RUN apk add <package-name>
RUN rm /var/cache/apk/*
```

In the above two examples, the preferred option is “—no-cache” as it delegates to the package manager the actual cleanup process and removes the user from that responsibility, possible typos or syntax error scenarios that would otherwise prevent the successful cleanup.

## Best Practice #5: Keeps the Layers to a Minimum

What layers are you talking about?

It is important to understand that Docker images are composed of layers.
When creating a Dockerfile, each new line of instruction is used to construct a separate layer during the image build process. This can quickly spiral out of control as shown below:

```
FROM containers.mycorp.com/alpine:tagName

RUN apk update
RUN apk add -—no-cache curl
RUN apk add -—no-cache nodejs
RUN apk add -—no-cache nginx
RUN apk add -—no-cache nginx-mod-http-geoip2
COPY index.html /var/www/html/
EXPOSE 80
CMD [“nginx”,“-g”,“daemon off;”]
```

There is a trick that allows you to optimize the above Dockerfile via the use of “&&” and “\” (i.e. chaining of statements) to reduce the number of layers created. Using the above example, the recommended alternative is as follows:

```
FROM containers.mycorp.com/alpine:tagName

RUN apk update && apk add –-no-cache curl nginx nginx-
mod-http-geoip2
COPY index.html /var/www/html/
EXPOSE 80
```

## Best Practice #6: Use .dockerignore

One quick and easy way to prevent unwanted files from being included into your image is via the use of the “.dockeignore” file. This can be useful for the following scenario:

```
FROM containers.mycorp.com/node:12.18.1

# instructs Docker to use this path as default location for all subsequent commands
WORKDIR /app
COPY package.json ./
# install all dependencies into “./node_modules” directory (inside image)
RUN npm install
COPY . .
EXPOSE 9090
CMD [ “node”, “index.js” ]
```

In the above example, the “npm install” command is already placing the latest dependency components into the image, namely the node_modules directory. However, the COPY command that follows is likely going to overwrite this node_modules directory with older content from local disk. You may add “node_modules” into the “.dockerignore” file to avoid this from happening during the Docker image build process.

Developer environments often contain SSH keys, .env files, SSL certificates, properties files that contain secrets in clear text, etc. The “.dockerignore” file is very useful for preventing the unwanted copy of sensitive data (from local disk).

Consider the following **“.dockerignore”** file as a reference:

```
.env
*pem
*.md
.metadata
.git
.github
.cache
.circleci
integration
img
node_modules
certs
keys
```

## Best Practice #7: Do Not Store Secrets In the Image

Use extreme caution with the contents of your Dockerfile. It allows for the declaration of environment variables with user-defined values. This feature can easily be misused. The ENV variable must not be used to define secrets, credentials or access tokens via the Dockerfile. Similarly, copying files with sensitive data into the image must be carefully prevented (i.e. including hidden files such as “.env” or similar). Once these secrets find their way into the image and the image is added to a repository with a large user base, the repository can now be used for harvesting secrets.

Embedded secrets are a high security risk for your company and may be instrumental for a successful data breach.

## Best Practice #8: Avoid Use of ADD (Favor Use of COPY)

Both ADD and COPY allow you to copy from local disk to the image location. However, ADD is more powerful as it supports remote URL locations for adding content at build time. In terms of security, it is best practice to get the content locally, scan for malware, verify it and then rely on COPY. This practice helps mitigate against the tidal wave of supply chain attacks that impacts all developers.

## Best Practice #9: Do Not Root, Do Not Sudo

With Docker, every image that does not specify a user will default to “root” (tragically). There are some vendor platforms that offer ways to prevent this, such as Red Hat OpenShift or Mirantis Docker Engine. Do not allow your image to run as “root”. This default “root” user in the container is the same “root” as on the host machine. This poses a great risk in the event of a container escape.
Similarly, **do not attempt to sudo** anywhere in the Dockerfile!

As a layer of defense (i.e. contain the blast radius), it is best practice to always define a user in the Dockerfile. This may be done as follows:

```
FROM containers.mycorp.com/alpine:tagName

# creates user home directory, ensures bash is the default shell
#
RUN useradd -ms /bin/bash runner
USER runner
WORKDIR /home/runner
COPY package.json ./
RUN npm install
CMD [“node”,“index.js”]
```

- Use the `-m` (`--create-home`) option to create the user home directory.
- Use the `-s` (`--shell`) option to specify the new user’s “login shell” (optional, not necessary for Production deployments where immutable Pod instances are strictly enforced).

A simpler way to define the new user may be as follows:

```
RUN useradd -ms --uid 10000 runner
USER 10000
```

**Note**: Every command that follows after “USER” (including interactive shell sessions) will execute as user **runner**. Interactive shell sessions in Kubernetes are only possible when the base OS image (that was chosen) contains shell command executables and the user has “kubectl” CLI access with the proper RBAC permissions to access the Pod.