Secrets
ABAP
Apex
AzureResourceManager
C
C#
C++
CloudFormation
COBOL
CSS
Dart
Docker
Flex
Go
HTML
Java
JavaScript
JCL
Kotlin
Kubernetes
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

Docker static code analysis

Unique rules to find Vulnerabilities, Security Hotspots, and Code Smells in your DOCKER code

All rules 44 | Vulnerability 4 | Bug 4 | Security Hotspot 15 | Code Smell 21

Tags | Impact | Clean code attribute | Search by name...

Allowing non-root users to modify resources copied to an image is security-sensitive
Security Hotspot

Automatically installing recommended packages is security-sensitive
Security Hotspot

Running containers as a privileged user is security-sensitive
Security Hotspot

Delivering code in production with debug features activated is security-sensitive
Security Hotspot

Use ADD instruction to retrieve remote resources
Code Smell

Arguments in long RUN instructions should be sorted
Code Smell

Track uses of "TODO" tags
Code Smell

Descriptive labels are mandatory
Code Smell

Use digest to pin versions of base images
Code Smell

Dockerfile parsing failure
Code Smell

Pulling an image based on its digest is security-sensitive
Security Hotspot

Running containers as a privileged user is security-sensitive

Analyze your code

Intentionality - Complete | Security

Security Hotspot | Minor | dockerfile cwe

Running containers as a privileged user weakens their runtime security, allowing any user whose code runs on the container to perform administrative actions.
In Linux containers, the privileged user is usually named `root`. In Windows containers, the equivalent is `ContainerAdministrator`.

A malicious user can run code on a system either thanks to actions that could be deemed legitimate - depending on internal business logic or operational management shells - or thanks to malicious actions. For example, with arbitrary code execution after exploiting a service that the container hosts.

Suppose the container is not hardened to prevent using a shell, interpreter, or Linux capabilities. In this case, the malicious user can read and exfiltrate any file (including Docker volumes), open new network connections, install malicious software, or, worse, break out of the container's isolation context by exploiting other components.

This means giving the possibility to attackers to steal important infrastructure files, intellectual property, or personal data.

Depending on the infrastructure's resilience, attackers may then extend their attack to other services, such as Kubernetes clusters or cloud providers, in order to maximize their reach.

Ask Yourself Whether

This container:

- Serves services accessible from the Internet.
- Does not require a privileged user to run.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

In the Dockerfile:

- Create a new default user and use it with the `USER` statement.
  - Some container maintainers create a specific user to be used without explicitly setting it as default, such as `postgresql` or `zookeeper`. It is recommended to use these users instead of root.
  - On Windows containers, the `ContainerUser` is available for this purpose.

Or, at launch time:

- Use the `user` argument when calling Docker or in the docker-compose file.
- Add fine-grained Linux capabilities to perform specific actions that require root privileges.

If this image is already explicitly set to launch with a non-privileged user, you can add it to the safe images list rule property of your SonarQube instance, without the tag.

Sensitive Code Example

For any image that does not provide a user by default, regardless of their underlying operating system:

```
# Sensitive
FROM alpine

ENTRYPOINT ["id"]
```

For multi-stage builds, the last stage is non-compliant if it does not contain the `USER` instruction with a non-root user:

```
FROM alpine AS builder
COPY Makefile ./src /
RUN make build
USER nonroot

# Sensitive, previous user settings are dropped
FROM alpine AS runtime
COPY --from=builder bin/production /app
ENTRYPOINT ["/app/production"]
```

Compliant Solution

For Linux-based images and scratch-based images that untar a Linux distribution:

```
FROM alpine

RUN addgroup -S nonroot \
    && adduser -S nonroot -G nonroot

USER nonroot

ENTRYPOINT ["id"]
```

For Windows-based images, you can use `ContainerUser` or create a new user:

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019

RUN net user /add nonroot

USER nonroot
```

For multi-stage builds, the non-root user should be on the last stage:

```
FROM alpine as builder
COPY Makefile ./src /
RUN make build

FROM alpine as runtime
RUN addgroup -S nonroot \
    && adduser -S nonroot -G nonroot
COPY --from=builder bin/production /app
USER nonroot
ENTRYPOINT ["/app/production"]
```

For images that use `scratch` as their base, it is not possible to add non-privileged users by default. To do this, add an additional build stage to add the group and user, and later copy `/etc/passwd`.

Here is an example that uses `adduser` in the first stage to generate a user and add it to the `/etc/passwd` file. In the next stage, this user is added by copying that file over from the previous stage:

```
FROM alpine:latest as security_provider
RUN addgroup -S nonroot \
    && adduser -S nonroot -G nonroot

FROM scratch as production
COPY --from=security_provider /etc/passwd /etc/passwd
USER nonroot
COPY production_binary /app
ENTRYPOINT ["/app/production_binary"]
```

See

- CWE - CWE-284 - Improper Access Control
- nginxinc/nginx-unprivileged: Example of a non-root container by default
- Microsoft docs, When to use ContainerAdmin and ContainerUser user accounts

Available In:

sonarlint | sonarcloud | sonarqube