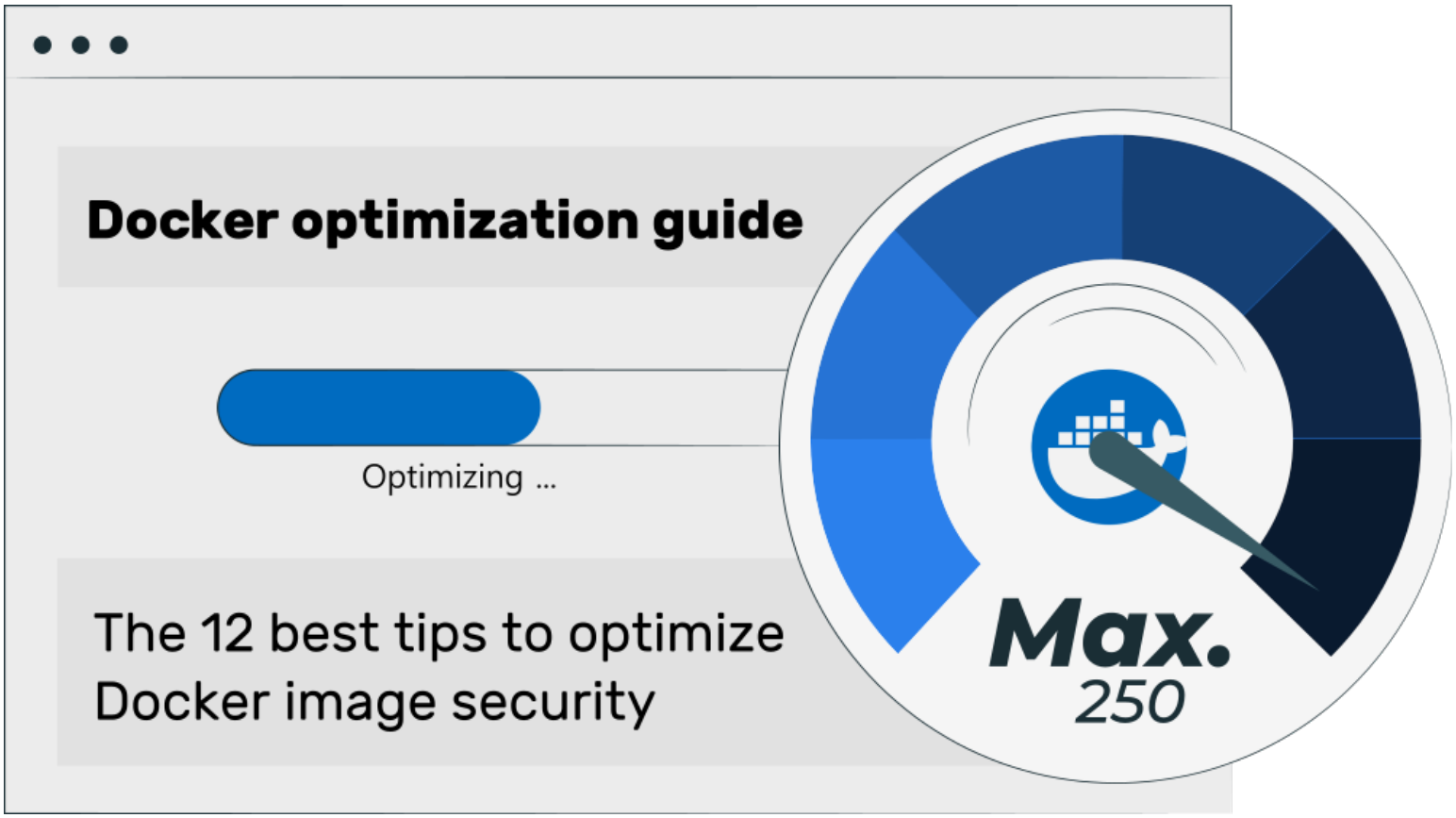

Docker optimization guide: the 12 best tips to optimize Docker image security

2022-02-20 by Marius

This article introduces 12 tips to optimize your Docker image security. For each tip, it explains the underlying attack vector, and one or more mitigation approaches. Tips include avoiding leaking of build secrets, running as non-root user, or how to make sure to use the most recent dependencies and updates.

Table Of Contents

Docker optimization guide series

This article is part of a multi-part series on working with Docker in an optimized way:

– Optimize Docker development speed
– Optimize Docker image build speed in CI
– Optimize Docker image size
– Optimize Docker image security (this article)

Introduction

When you are new to Docker, you will most likely create *insecure* Docker images that make it easy for attackers to take over the container, or possibly even the entire host, which then allows the attacker to infiltrate other infrastructure of your company.

There are many different attack vectors that can be abused to take over your system, such as:

- The started application (specified in the `ENTRYPOINT` of your `Dockerfile`) runs as `root` user. Consequently, once an attacker has exploited a vulnerability and gains shell access, they can take over the *host* on which the Docker daemon is running.
- Your image is based on an outdated and/or insecure *base* image, which contains security exploits which are (now) well-known.
- Your image contains tools (such as `curl`, `apt`, etc.) that allow an attacker to load further malware into the container, once they have gained some kind of access.

The following sections explain different approaches to optimize your image

security. They are sorted by importance/impact, such that the more important ones are listed first.

1. Avoid leaking build secrets

***Build secrets* are credentials that are only needed while *building* your Docker image (not at run-time).** For instance, you might want to include a compiled version of some application into your image whose source code is closed-source, and its Git repo is access-protected. While building the image, you need to clone the Git repo (which requires the build secrets, e.g. SSH access keys to that repo), build the application from source, and then delete the sources (and secrets) again.

"Leaking" a build secret means that you accidentally baked such secrets into one of the layers of your image. This is bad, because *anyone* who pulls your image can retrieve the credentials. The problem originates from the fact that Docker images are built layer by layer, in a purely *additive* way. **Files you delete in a layer are only *marked* as deleted, but can still be accessed by everyone pulling your image, using advanced tooling.**

Use one of the following two approaches to avoid leaking build secrets:

A. Multi-stage builds

Docker Multi-stage builds (official docs) have many different use cases, e.g. *speeding up* your image build, or *reducing the image size*. Other articles of this series go into details regarding these other use cases. Anyways, **you can also multi-stage builds to avoid leaking build secrets, as follows:**

- **Create a stage #A into which you** `COPY` **the credentials and use them to retrieve other artifacts** (e.g. the Git repo of the above example), **and perform further steps with them** (e.g. compiling an application). **The stage #A build *does* contain the build secrets!**
- **Create a stage #B into which you copy only non-secret artefacts from stage #A**, such as a compiled application.

- **Only publish/push the stage #B image**

B. BuildKit secrets

Background info

If you use `docker build` for building, there are multiple *backends* that actually perform the build. The newer and faster backend is BuildKit, which you need to be explicitly enable on *Linux* by setting the environment variable `DOCKER_BUILDKIT=1`. Note that BuildKit is enabled by default on Docker for Desktop on Windows/macOS.

As explained in the docs here (read them for more details), the BuildKit build engine supports additional syntax in the `Dockerfile`. **To use a build secret, put something like this in your** `Dockerfile`**:**

```
RUN --mount=type=secret,id=mysecret,dst=/foobar <command to run>
```

This makes secrets available to the build container while that `RUN` **statement is executed, but does *not* put the secret itself** (here: the `/foobar` folder) **into the built image**. **You need to specify the path to the secret's *source* file/folder (located on the host) when running the** `docker build` **command**, e.g.

```
docker build --secret id=mysecret,src=mysecret.txt -t sometag .
```

There is one caveat, however: **you cannot build images that require secrets via** `docker-compose up --build`, because Docker-compose does not support the `--secret` argument for building yet, see GitHub issue. If you rely on docker-compose builds to work, use approach 1 (Multi-stage builds) instead.

Side note: do not push images built on a development machine

You should always build and push images in a *clean* environment, e.g. a CI/CD

pipeline, where the build agent clones your repository into a *new* directory.

The problem with using your *local development machine* for building is that your local "working tree" of the Git repository might be *dirty*. For instance, it might contain files with secrets that you need during development, e.g. access keys to staging or even production servers. If these files are not excluded via `.dockerignore`, a statement such as "`COPY . .`" in your `Dockerfile` could accidentally lead to leaking these secrets into the final image.

2. Run as non-root user

By default, when someones runs your image via "`docker run <more arguments> yourImage:yourTag`", the container (and the programs you have in your `ENTRYPOINT` / `CMD`) runs as `root` user (in the container and on the host). This gives an attacker, who gained shell access in your running container using some exploit, the following powers:

- *Unrestricted* write-access (due to being `root`) to all those directories on the host that are explicitly mounted into the container.
- Ability to do everything in the container that a Linux root user can do. For instance, an attacker could install additional tools they need to load even more malware, e.g. via `apt-get install` (a non-root user could not do this).
- *If* the container of your image was started with `docker run --privileged`, the attacker can even take over the entire *host*.

To avoid this, **you should run your application as non-root user**, that is, some user that you created during the `docker build` process. **Place the following statements in your `Dockerfile` somewhere (usually towards the end):**

```
# Create a new user (including a home-directory, which is optional)
RUN useradd --create-home appuser
# Switch to this user
USER appuser
```

All commands in the `Dockerfile` that come *after* the `USER appuser` statement (e.g. `RUN`, `CMD`, or `ENTRYPOINT`) **will be run with this user. There are a few caveats to be aware of:**

- **Files you copied into your image via `COPY`** (or files created by some `RUN` commands) ***before* switching to the non-root user are owned by `root`, and are consequently not *writable* by your application running as non-root.** To fix this problem, move the code that creates and switches to the non-root user closer to the beginning of the `Dockerfile`.
- **Files that your program expects to be somewhere in the user's *home* directory (e.g. `~/.cache`) might now suddenly be missing from your app's perspective**, if these files were created at the beginning of the `Dockerfile`, as `root` user (being stored below `/root/` and not below `/home/appuser/`).
- **If your application listens to a TCP/UDP port, your app must use ports > 1024**. Ports <= 1024 can only be used either as `root` user, or with high Linux capabilities, which you should not give to your container just for that purpose.

3. Use the latest base image build & update system packages

If you are using a base image that contains the entire toolset of a real Linux distribution (such as Debian, Ubuntu or alpine images), **including a *package manager*, it is recommended to use that package manager to install all available package updates.**

Background

Base images are maintained by someone who configured scheduled CI/CD pipelines that build the base image and push it to Docker Hub in regular intervals. You have no control over this interval, and it often happens that security patches are available in the Linux distro's *package registry* (e.g. via `apt`) *before* that pipeline pushes an updated Docker image to Docker Hub. For instance, even if a base image is pushed once per *week*, it could still happen that security updates are available a few *hours*

or days after the most recent image was published.

Therefore, it's a good idea to always run package manager commands that update the local package database and install updates, in *unattended mode*, which does not require user confirmation. **The command differs for each Linux distribution.**

For instance, **for Ubuntu, Debian, or derivative distros**, use `RUN apt-get update && apt-get -y upgrade`

Another important detail is that you need to tell Docker (or whatever image build tool you use) **to refresh the base image**. Otherwise, **if you reference a base image such as** `python:3` **(and Docker already has such a image in its *local* image cache), Docker won't even check whether a newer version of** `python:3` **exists on Docker Hub. To get rid of this behavior, you should use this command:**

```
docker build --pull <rest of the build command>
```

This makes sure that Docker will pull updates of the image(s) mentioned in the `FROM` **statement(s) of your** `Dockerfile`**, prior to building your image.**

You should also be aware of Docker's *layer caching* mechanism, which causes your image to become stale, because the layer for the `RUN <install apt/etc. updates>` **command is cached, until the base image maintainer releases a new version of the base image. If you find out that the release frequency of base image is rather low** (e.g. less often than a week), **it is a good idea to regularly** (e.g. once per week) **rebuild your image with disabled layer caching. You can do so by running the following command:**

```
docker build --pull --no-cache <rest of the build command>
```

4. Regularly update third party dependencies

The **software you write is based on *third party dependencies***, meaning software made by other people. This includes:

- The base Docker image your image is based on, or
- Third party software components you use as part of your application, e.g. installed via `pip`/`npm`/`gradle`/`apt`/...

Once these dependencies become outdated in your image, this increases the attack surface again, because outdated dependencies often have exploitable security vulnerabilities.

You solve this problem by regularly using SCA tools (Software Composition Analysis), such as Renovate Bot. These tools (semi-) automatically update your declared third party dependencies to their most recent version, e.g. in your `Dockerfile`, Python's `requirements.txt`, NPM's `packages.json`, etc. You need to design your CI pipelines such that the change made by the SCA tool automatically triggers a re-build of your image.

Such automatically-triggered image rebuilds are particularly useful for projects which are in maintenance-only mode, but where the code shall still be used in production by customers (who expect it to be secure). During the maintenance period, you are no longer developing new features, and no new images would be built, because there are no new commits (made by you) triggering new builds. However, the commits made by the SCA tool do trigger the image builds again.

You can find more details about Renovate bot in my related blog post.

5. Have your image scanned for vulnerabilities

Even if you implemented the above advice, such that your images always use the *latest* third party dependencies, it can still be *insecure* (e.g. if a dependency has become abandoned). In this context, "insecure" means that one (or more) of the dependencies have *known* security vulnerabilities (registered in some

CVE database).

For this reason, **there are various tools that you provide your Docker image, and they scan through all contained files to find such vulnerabilities. These tools come in two forms:**

1. **CLI tools that you explicitly invoke, e.g. in a CI pipeline**, e.g. Trivy (OSS which is very easy to use in a CI pipeline, see Trivy docs), Clair (OSS, but more complex to set up and use than Trivy), or Snyk (integrated into the Docker CLI via "`docker scan`", see cheat sheet, but there is only a *limited* free plan!)
2. **Scanners integrated into the image registry that you push your image into**, e.g. Harbor (which uses Clair or Trivy internally). There are also commercial offerings such as Anchore.

Because these scanners are generic and try to cover a broad range of package registries, they might not be particularly specialized for the programming language or package registries you use in your project. It can sometimes make sense to investigate which tools your programming language ecosystem offers. For instance, for Python there is the safety tool which is specialized on Python packages.

6. Scan your `Dockerfile` for violations against best practices

Sometimes problems arise from statements you place in your `Dockerfile`, which are bad practice (without you realizing it). Use tools such as checkov, Conftest, trivy, or hadolint, which are linters for `Dockerfiles`. To make the right choice for the tool, **review** which **default rules/policies** are shipped with it. **For instance,** `hadolint` **offers many more rules than** `checkov` **or** `conftest`**, because it is specialized for** `Dockerfiles`**.** The tools also complement each other, therefore it does make sense to run multiple tools, e.g. hadolint and trivy, on your `Dockerfiles`. **Be prepared, though, that you need to maintain "ignore files" where certain rules are ignored, e.g. due to false positives, or because you are deliberately**

breaking a rule.

7. Do **NOT** use Docker Content Trust for Docker Hub

Docker Content Trust **(as explained in the official docs) is a feature to verify that pulled (base) images are *really* built & pushed by the company or organization behind that image. The feature can be enabled by setting the** `DOCKER_CONTENT_TRUST` **environment variable to** `"1"` **while running** `docker build` **or** `docker pull`**. The Docker daemon will refuse pulling images that have not been signed by the publisher.**

Unfortunately, the community stopped signing images this way about a year ago. Even Docker Inc. stopped signing the official Docker image in December 2020, with no official explanation. What is even more problematic is that if you do something like "`docker pull docker:latest`**", an image will be downloaded, but it might be quite out of date.**

There are alternative implementations for image signing (which I have not checked out yet), such as cosign.

8. Scan your own code for security issues

Security issues usually arise from issues with *other people's* code, that is, popular third party dependencies which are "lucrative" to hack because they are wide-spread. However, **sometimes it is *your own* code that is to blame. For instance, you might have accidentally implemented SQL inject possibilities, stack overflow bugs, etc.**

To find those issues, there are so-called SAST tools (Static Application Security Testing). One the one hand, there are programming-language-specific tools (which you have to research individually), such as bandit for Python, or Checkstyle / Spotbugs for Java. **On the other hand there are tool suites (some of which are non-free/commercial) that support *multiple* programming languages and**

frameworks, such as SonarQube (for which there is also the SonarLint IDE plugin). **See here for a list of SAST tools.**

There are two basic approaches for doing security scans in practice:

1. **Continuous (automatic) scanning**: you create a *CI job* that scans your code on each push. This constantly keeps the security of your code on a high level, but you

2. **Occasional (manual) scanning**: some security-minded member of your team runs the security check locally, e.g. once per month or before every release, and manually looks over the result.

9. Use `docker-slim` to remove unnecessary files

The docker-slim tool takes large Docker images, runs them temporarily, analyzes which files are really used in the temporary container, and then produces a new, *single*-layer Docker image where all unused files have been removed. This has two benefits:

1. **The image size is reduced**
2. **The image becomes more secure**, because tools are removed that are not needed (e.g. `curl`, or the package manager)

Please refer to the *Docker slim* section in my previous article for further details.

10. Use *minimal* base images

The more software (e.g. CLI tools, etc.) is stored in an image, the larger the attack surface becomes. It's good practice to use "minimal" images, which are as *small* in size as possible (which is a good advantage anyway), and contain as few tools as possible. **Minimal images go even beyond what "size-optimized" images (such as `alpine` or `<something>:<version>-slim`, e.g. `python:3.8-slim`) do: they come without any**

package manager. This makes it hard for an attacker to load additional tools.

The most secure minimal base image is `SCRATCH`, which contains absolutely nothing. Starting your `Dockerfile` with `FROM SCRATCH` is only feasible if you are placing self-contained binaries in the image, that have all dependencies (including C-runtimes) baked in.

If `SCRATCH` does not work for you, Google's distroless image can be a good alternative, especially if you are building applications for common programming languages, such as Python or Node.js, or need a minimal base image of *Debian*.

Unfortunately, minimal images have several caveats to be aware of:

- **Caveats of distroless:**
  - Using the programming-language-specific images published by Google on `gcr.io` is *not* recommended, because there is only a `latest` version tag, as well as tags for *major* versions (e.g. "3" for python, or "12" for Node). You have no control over the specific language run-time versions (e.g. whether Python 3.8.3 or 3.8.4 etc. is used), which breaks the reproducibility of your image builds.
  - Customizing (and building your own) distroless images is quite involved: you need to get acquainted with the Bazel build system and build the images yourself
    - Note: if the only customization you need is to run code as *non-root* user, there is a `nonroot` user by default in every distroless base image, see here for details.
- **Caveats of minimal base images in general:**
  - Debugging containers using your minimal base images is tricky, because useful tools (such as `/bin/sh`) are now missing
    - For Docker, you can run a second debugging-container (that does have a shell and debugging tools, e.g. `alpine:latest`) and make it share the *PID namespace* of your minimal container, e.g. via `docker run -it --rm --pid=container:<minimal-container-id> --cap-add SYS_PTRACE alpine`

sh

- For Kubernetes, you can use *ephemeral containers*, see e.g. here

11. Use *trusted* base images

A *trusted* image is one that has been *audited* by someone (either your own organization, or someone else), e.g. with a security level. This can be particularly important to regulated industries (banking, aerospace, etc.) **with high security requirements and regulations.**

While the auditing could be done by yourself, by building trusted images yourself from scratch, this is discouraged, because you, the image builder, has to ensure that all auditing-related tasks are done and properly documented (e.g. documenting the list of packages in the image, executed CVE-checks and their results, etc.). This is a lot of work. Instead, **it is recommended to outsource this task, using *commercial* "trusted registries", which offer a selected set of trusted images, e.g. RedHat's Universal Base Images (UBI). RedHat's UBIs are now also available on Docker Hub for free.**

Background

Images hosted on Docker Hub are not audited. They are provided "as-is". They might be insecure (even contain malware), and no one will notify you about it. Using an insecure base images from Docker Hub images will therefore also make *your* image insecure.

Also, you should not confuse auditing with Docker's *content trust*, mentioned above! Content trust only confirms the identity of the source (the image uploader), and does not state any facts about the image security.

12. Test whether your image works with reduced

capabilities

Linux *capabilities* are a Linux kernel feature that allow you to control which kernel features an application may use. Examples are whether a process may send signals (e.g. `SIGKILL`), **configure network interfaces, mount a disk, or debug processes. See here for the complete list. In general, the fewer capabilities your application needs, the better.**

Anyone who starts a container of your image can give (or take away) these capabilities, e.g. with a call such as "`docker run --cap-drop=ALL <image>`". **By default, Docker drops all capabilities except for those defined here. Your application might not need all of them.**

As a best practice, try starting a container of your image, dropping *all* capabilities (using `--cap-drop=ALL`) **and see whether it still works. If it does not, figure which capabilities are missing, whether you really need them, and if you do, document which capabilities your image needs (and why), which increases trust with whoever runs your image.**

Conclusion

Making your image secure is no walk in the park. It takes time to evaluate and implement each practice. The list in this article should save you time, because it already did the work of collecting and prioritizing the necessary steps.

Fortunately, securing your application is an *iterative* process. You can start small, and implement one step at a time. You do need buy-in from management, though. This is sometimes tricky, especially if your manager is resistant to advice, and if they tend to extrapolate from past experiences (*"we, or our customers, have never been hacked before, so why should this happen to us now? We need features instead!"*). There are several options you have to convince your manager to allocate resources to security. For instance, if you have a direct channel to your customers (for which you build the software), convince *them* that they need security, so that they ask for security as a

"feature". Alternatively, find reports of security breaches in your industry (e.g. where a direct competitor was affected), to demonstrate that hacking attacks *do* actually happen, even in your industry, and that they have severe (financial) repercussions.

Do you know any further tips for strengthening the image security? Let me know in the comments!

Docker, Development

‹ Docker optimization guide: 8 tricks to optimize your Docker image size

Leave a Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment