snyk | docker

Guide to Container Security For Development Teams

A joint collaboration between Docker and Snyk

Three steps to creating a secure container image

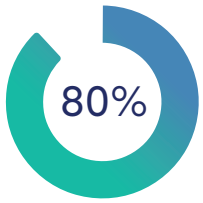
Container security is not a single area of concern – it spans developers, security, and operations teams. There are an array of security layers that apply to containers:

- **the container image itself and the software inside,**
- **the interaction between a container, the host operating system, and other containers on the same host**
- **the host operating system itself,**
- **container networking and storage concerns,**
- **and security at runtime, often in Kubernetes clusters.**

1. Secure your code and its dependencies

Delivering your applications faster is likely one of the key reasons you're creating a container in the first place and is the lifeblood of your organization. In the not-too-distant past, application security began and ended with the code and while containers and other modern development practices have expanded the broad meaning of "application code," this particular area of concern still remains.

Fortunately, this is the portion of container images that's most directly controlled by developers and, hopefully, the best understood. Nonetheless, tracking down all your code dependencies and figuring out how to fix security issues isn't trivial. Assuming you have access to the source code itself, you should use purpose-built tools - like Snyk Open Source - to do software composition analysis (SCA) and static application security testing (SAST) to analyze your code and its dependencies. In modern applications, it's not unusual for the 3rd party open source dependencies to make up the majority of the lines of code in an application.



of application code is open source

of open source vulnerabilities are found in transitive dependencies.

Figure 4: The amount of open source code in an app can easily dwarf the amount of proprietary code.

Spotting issues early in development and integrating tools with your source code opens up the possibility to automate this process, independent of the containerization process. It is possible to scan a container and analyze some types of code, however, catching these issues directly in your git commits and repositories will likely fit a developer's process better.

2. Start with a minimal base image from a trusted source

What's the big deal about small images?

The base image – the `FROM` line in your Dockerfile - is one of the most important considerations when it comes to security. Fortunately, many trustworthy vendors provide content you can easily use. Docker Hub is by far the most popular starting point for sourcing container base images.

Docker Hub has more than 3.8 million available images and more than 7 million repositories. Docker Hub is very active and sees about 11 billion pulls per month. Some of these images are Official Images, which are published by Docker as a curated set of Docker open source and "drop-in" solution repositories.

Docker also offers images that are published by Verified Publishers. These high-quality images are published and maintained directly by a commercial entity whom Docker verifies as a Verified Publisher. Docker's guidelines for these verified publishers to follow are a great starting point for defining your own internal container image best practices.

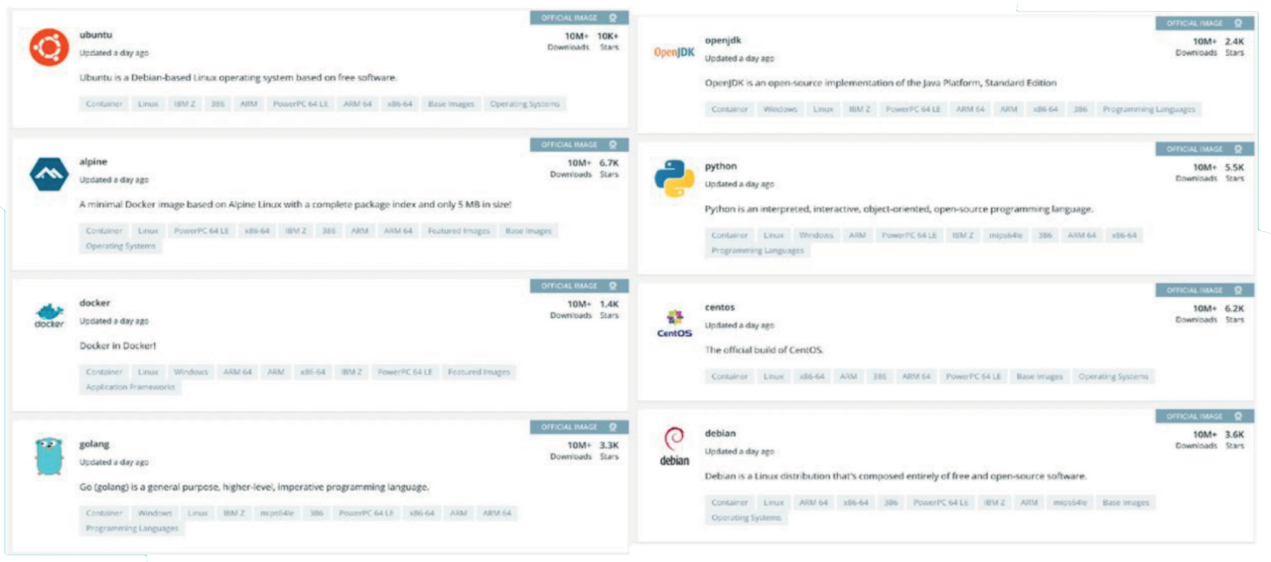


Figure 5: Official images on Docker Hub. Note the "Official" badge and recent updates.

It's easy to go to Docker Hub and find a publicly available image that matches your use case, but you need to pay attention to the provenance of the images you choose. Just like you wouldn't download and install software from an untrusted website, you likely would not want to use images pushed to Hub by users you don't know and trust.

By using images that are part of Docker's Official program, or if you know and can verify the source and contents of 3rd party images - perhaps using something like Notary to check digital signatures - then you have some level of assurance of quality. But to further reduce the number of vulnerabilities and add more control to what is packaged inside your containers you should go a step further and choose minimal base images matched to your needs.

As an example, in figure 5 above there is a Python repository shown, and you can certainly build your Python app and it will almost certainly work. That's because the image tagged on Docker Hub is designed to be easy to work with across many different use cases and it's well maintained. But there are more than 1000 other images in this registry.

Should you just use whatever comes in the easy-to-remember `python` *or are there smaller images that would suit your needs and also reduce your security footprint?* The answer, as you might surmise, is that there are almost certainly better choices from a security perspective.

Container image size matters and not just for portability and fast downloads. The image tagged `python` is simple to use because it comes with a fairly large footprint of pre-installed operating system libraries and developer packages. That means it will likely work great with a range of projects and will have everything you might need to compile code and dependencies; but vulnerability scanners might produce a long list of issues to track down, too.

3. Manage all the layers in between the base image and your code

We went into some depth on the base images because they require some special considerations. You inherit whatever comes in the base image as you build up your own image on top of it, while a slim image reduces the security burden. But what about all those layers you add to the container? If you start with a slim image, chances are you'll need to add tools and libraries, plus you'll have your code and various things to install to make things work and all of those need to be monitored for vulnerabilities.

The good news is you directly control these middle layers, which we consider to be everything after the first `FROM` line and the final Dockerfile lines where you set up your code to run. More specifically, we're interested in the `RUN` , `COPY` and `ADD` commands in Dockerfiles as these are the ones that will install things. Technically, your code might be somewhere in these middle layers, too, but philosophically we're going to call your code the final layer, mainly because we already dealt with the code in Step 1.

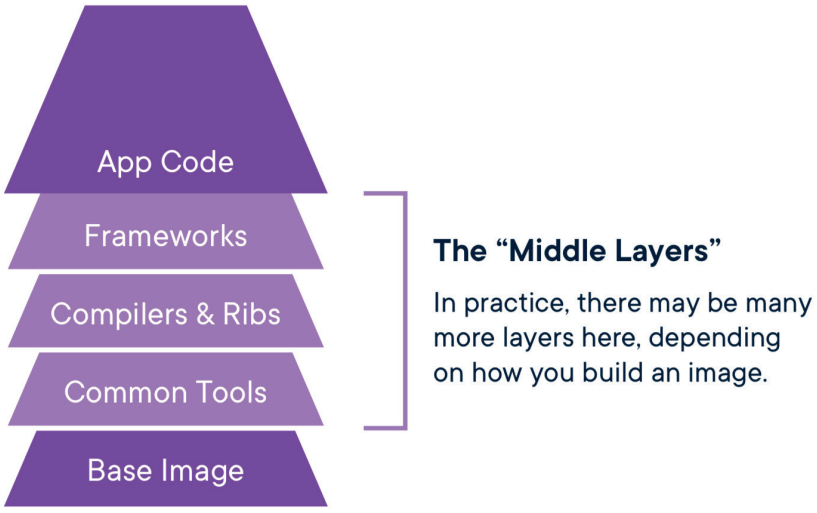


Figure 7: The middle layers of a container image

> **One of the most difficult things about managing vulnerabilities in these middle layers is prioritizing what to pay attention to in the various stages of the lifecycle.**

At each stage you might need different sets of tools, but as images head to production you should remove everything that isn't absolutely necessary. Customizing your images by starting with a minimal base and then adding your tools makes it very easy to remove these tools later by simply taking them out of the Dockerfile and rebuilding, or even better, by using multi-stage builds to capture all these stages in a single, automated build process.