Shifting Docker security left

snyk

Table of contents

Authors

Brian Vermeer (@BrianVerm)
Snyk

William Henry (@ipbabble)
RedHat

TL;DR

Docker vulnerabilities are widespread

- The top 10 official Docker images with more than 10 million downloads each contain at least 30 vulnerabilities.
- Of the top 10 most popular free certified Docker images, 50% have known vulnerabilities.

There is a lack of clear ownership for Docker security

- 68% believe developers should be responsible for owning container security, followed by operations (39%).
- 80% of developers say they don't test their Docker images during development.
- 50% of developers don't scan their Docker images for vulnerabilities at all.

Easy fixes can have big security impact

- Each of the top ten most popular default 20% of the Docker images with vulnerabilities could have been solved by a simple image rebuild.
- 44% of Docker images had known vulnerabilities for which there were newer and more secure base images available.
- 45% of developers never discover new vulnerabilities disclosed in their production containers.

How you can improve:

Choose the right Docker base image

- Work with the smallest base image: don't package what you don't need!
- Proactive remediation advice: tools such as Snyk offer recommended image upgrades and alternatives with the fewest vulnerabilities from which you can choose.

Scan your Docker images frequently

- Scan your Docker images as part of your development workflow and in your CI/CD pipelines.
- Proactive remediation advice: tools such as Snyk pinpoint which image layers introduce vulnerabilities and offer remediation advice.
- Monitor your Docker containers in production by automatically scanning your base image and packages.

Use multi-stage builds and build often!

- By using multi-stage builds, you only copy the artifacts you actually need, reducing unnecessary vulnerabilities in your images.
- Re-build your Docker images as part of your development pipeline.

An introduction to this report

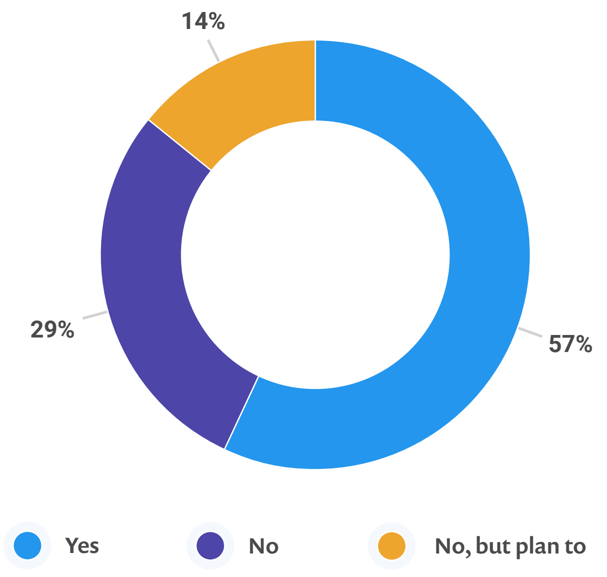
Docker turned 6 years old this year. Over the course of these 6 years, the adoption of Docker—and containers in general—has grown massively. Currently 57% of developers claim to use containers in their day-to-day work (see our graph later in this section). With the advent of container technologies, developers have been able to build innovative solutions more easily. Lots of these open source solutions are freely available. Today, even major software vendors post pre-built images in public registries such as Docker Hub, enabling developers to download and use these images.

In 2015 Docker, Red Hat, CoreOS, IBM, Google and a broad coalition of industry leaders focused on common standards for software containers, founding the Open Container Initiative (OCI). This initiative was founded with the goal to host an open source, technical community. In addition, it aims to build a vendor-neutral, open-specification and portable runtime for container-based solutions. Since the time OCI standards were released, many other build and runtime technologies have been developed. So, while this article often mentions "Docker containers" and "Docker images" it should be understood that there are many technologies available that adhere to the OCI (open container initiative) standards. At the time of writing this piece, Docker is still, by far, the most popular OCI build tool and runtime. The dominance of Kubernetes and its use of CRI-O is beginning to change that landscape.

As more organizations create, spread and use Docker containers, the risk of security vulnerabilities grows. Docker images are largely built on top of other images, meaning a vulnerability in one image is also present in all the images that utilize it.
The wide adoption of Docker comes at a price — a single vulnerability can be widely spread and have major impact.

We took a closer look at the raw data from our State of Open Source Security report, to gain a better understanding of the Docker landscape and its associated security risks and best practices. In this report we look at common security problems with Docker containers and Docker images and **some of the actions you can take** to improve security.

Do you use containers?



State of Docker security

The Docker landscape

Today, Docker is widely used and Docker Hub is the default place to go if you need a Docker image. Docker Hub is the world's largest library and community for container images. At the time of writing this blog, Docker Hub already had over 2 million (2,085,422) images available to download and use instantly, and that number grows daily.

Anyone can push images to Docker Hub but Docker Hub labels certain images as more trustworthy than others. While Docker offers official recognition and even certification for some contributors and their images, even those images still contain vulnerabilities that should be managed.

Currently, Docker Hub contains:

- **223** images published by verified publishers. These products are published and maintained directly by a commercial entity.
- **151** official images available for use. Official images are a curated set of Docker open source and "drop-in" solution repositories.
- **40** Docker images certified by Docker, meaning that they must conform to best practices and pass certain baseline tests.
- Each of the **top 10** official Docker images with more than **10 million downloads** and each of them with **at least 30 vulnerabilities**.
- Of the top 10 most popular free Docker certified images, half with known vulnerabilities. One image we checked had almost 160 known vulnerabilities.

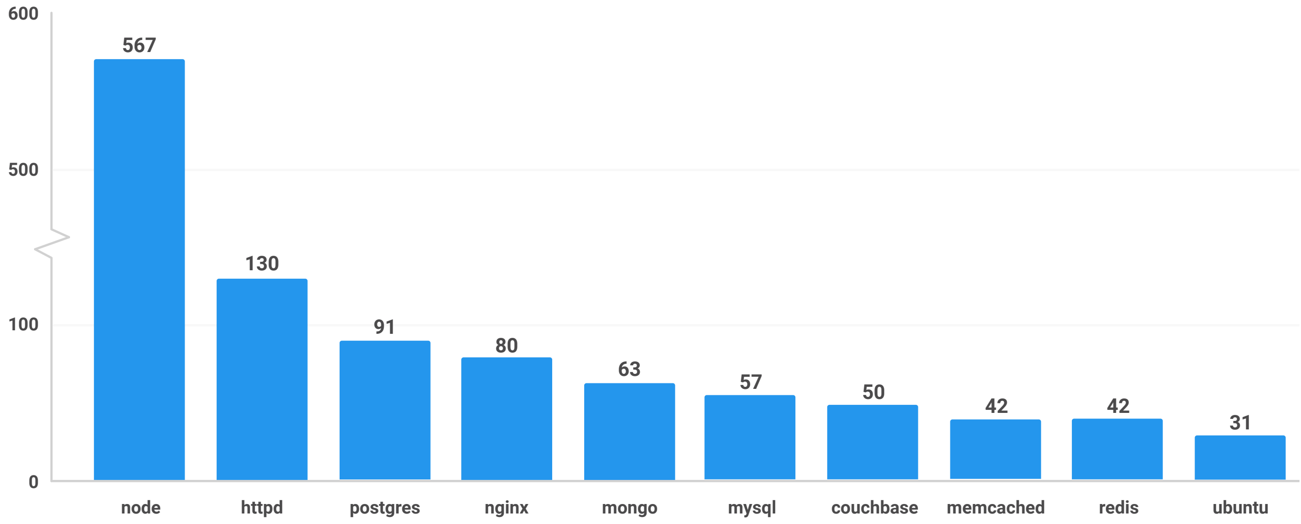
The adoption of application container technology is increasing at a remarkable rate and is expected to grow by a further 40% in 2020, according to 451 Research.

Known vulnerabilities in Docker images

Key takeaways

Each of the top 10 official images on Docker Hub contain at least 30 vulnerabilities

Vulnerabilities per Docker image



Docker Hub is the main source for publicly available Docker images. While Docker advises you to use official images or Docker-certified images as a security best practice, it can be seen that the top 10 most popular Docker images each contains vulnerabilities. All of these images are official images.

Accordingly, we decided to scan through ten of the most popular images with Snyk's recently released container vulnerability management features.

For every Docker image that we scanned, we were able to find vulnerable versions of system libraries. The last scan as of March 11, 2019 shows that the official Node.js image ships with 567 vulnerable system libraries. The remaining nine images ship with at least 31 publicly known vulnerabilities each.

Solutions

In light of these facts, following are container vulnerability management solutions that we suggest:

- Scanning images during development
- Scanning containers in production

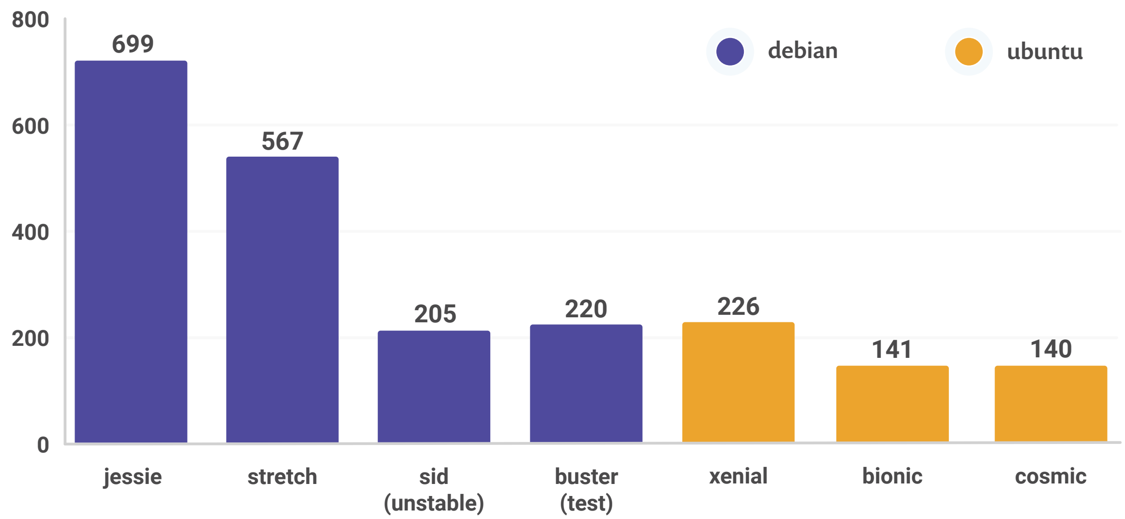
Vulnerabilities in base images

The majority of vulnerabilities are found in the operating system (OS) layer. The images described in the previous section are images that are built on top of a base image. **Therefore, the choice of a good base image is crucial in decreasing the number of vulnerabilities.**

The node image is built on top of one of the buildpack-deps images. The Docker buildpack-deps are a collection of common build dependencies used for installing various modules and widely used as a base image for building other images.

Currently, the default buildpack-deps version is “stretch”, which refers to the Linux distribution (distro) on which it is based. This stretch version contains 567 vulnerabilities—-corresponding precisely to the number of vulnerabilities in the latest node image that uses this buildpack-deps image as its base image. It is striking that the three buildpacks that are based on ubuntu images (xenial, biomic and cosmic) contain fewer vulnerabilities than the debian-based buildpacks, suggesting that currently ubuntu-based images are a better choice from a security standpoint.

Vulnerabilities in buildpack-deps



Key takeaways

The top two most popular base images each have over 500 vulnerabilities

Solutions

In light of these facts, following are container vulnerability management solutions that we suggest:

- Choosing the right base image
- Rebuilding images

Addressing Docker security

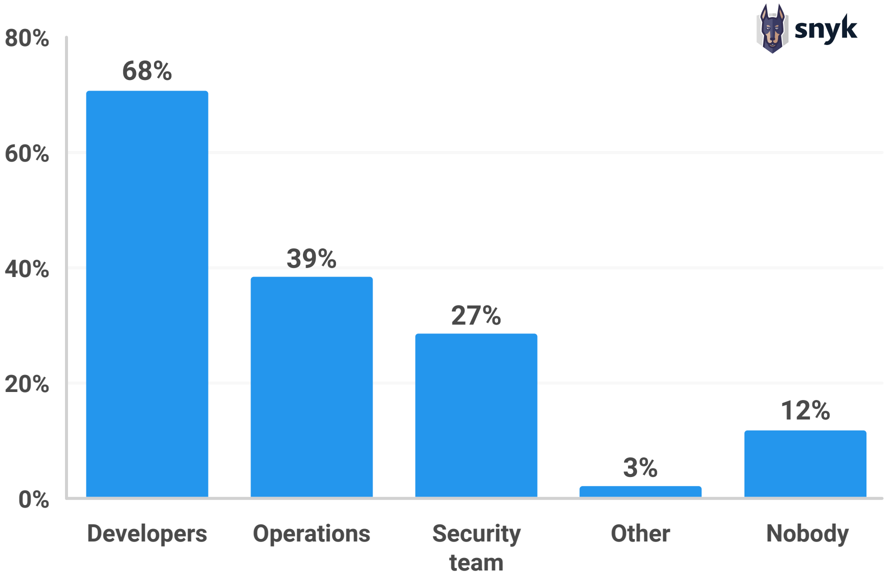
Container security ownership

When Docker is an integral part of your own ecosystem, somebody needs to take responsibility for it. Most people agree that responsibility is a shared effort. When asked who owns container security (with the option to select more than one answer) most respondents believed that developers should be responsible for owning container security (68%), thereafter followed by operations (39%).

Although security ownership is a shared effort within DevOps, or better yet—DevSecOps, we see that the majority believe that developers play a key role. This is similar to what we see regarding the security responsibility for application code. According to The State of Open Source Security report, 81% of respondents believe that developers should own the security of their applications.

Organizations may want to also consider partnering with the trusted vendor of the base images in order to manage some of the security risk when working with your CI/CD pipelines. Vendors such as Red Hat provide scanned and signed images. They also provide fast turnaround to CVEs on base images that can then trigger application image rebuilds in CI/CD pipelines. This doesn't negate the need for scanning during the DevOps lifecycle but reduces the latency between vulnerable production images and healthy production images. Sharing the risk with a third party can help reduce the overall risk.

Who is responsible for your container image security?



Key takeaways

- 68% of developers believe they themselves should be responsible for container security.
- 50% of developers don't scan their operating system (OS) layer Docker image for vulnerabilities.
- 45% of developers never discover new vulnerabilities disclosed in their production containers.
- 80% of developers don't test their Docker images during development.

Security testing the OS layer

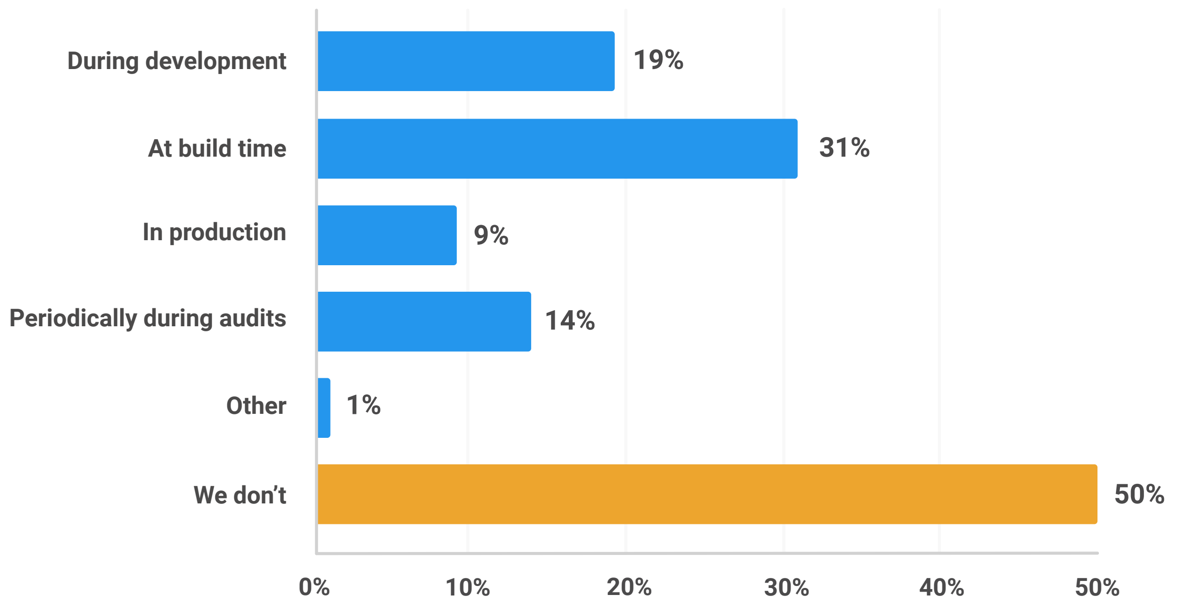
Unfortunately, just 19% of developers claim to test their Docker images during development for vulnerabilities in the Operating System layer. This means that over 80% of developers do not shift security left to test their images during development—unfortunately, discovering vulnerabilities later is costlier.

On top of this, half (50%) of the users don't perform any sort of scan for the OS layer of their Docker image. It's important to understand what is in the OS layer of your images. Blindly using Docker images is very dangerous as you'll no doubt bring in countless vulnerabilities. You can easily reduce the risk by scanning them and first understanding the known vulnerabilities in each image you're considering.

By using a scanning tool for Docker images, such as the one Snyk provides, the vulnerable images can be caught throughout the complete development cycle. Starting locally, when a developer selects a particular Docker image, scanning prevents vulnerable Docker images coming into your environment. Repeat scanning in all stages and through the end of the CI/CD pipeline to prevent an image with known vulnerabilities from going into production.

Vulnerabilities in images are naturally discovered as time passes, making it crucial to keep track of your production images. The median time before a vulnerability is found and reported is 2.5 years. Currently, 9% of users claim to scan their images when in production. An image might have been safe when it was deployed in production, but from that point forward over 90% of developers wouldn't know the current security status of their images.

When do you scan your Docker image for OS vulns?



Finding out about vulnerabilities

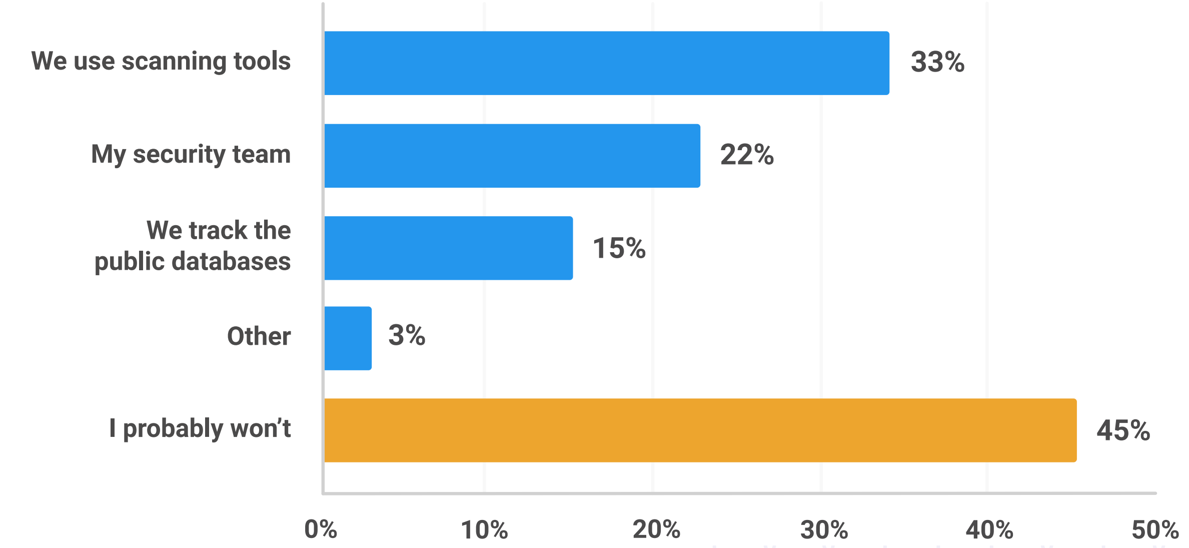
We also asked how people find out that a container they are running contains disclosed vulnerabilities. 45% probably never check. A reasonable amount (15%) of the respondents check public databases such as the CVE database. Keep in mind that the CVE database is only the tip of the iceberg though. The process of registering a CVE may take several weeks and many vulnerabilities might not even become a CVE entry, but may still be a major risk to your system.

Solutions

In light of these facts, following are container vulnerability management solutions that we suggest:

- Use multi-stage builds
- Rebuilding images
- Scanning images during development
- Scanning containers in production

How do you find out about new vulnerabilities in your deployed containers?



How to improve your Docker security

There are many actions we can take to lower the risk of running in a container that someone might exploit. These include:

- Choosing the right base image
- Use multi-stage builds
- Rebuilding images
- Scanning images during development
- Scanning containers in production

We also have a Docker security cheat sheet containing best practices that you can follow.

Key takeaways

- Work with the smallest base image: Don't package what you don't need!
- All 567 vulnerabilities in the Node.js image are inherited from the base image.

Choosing the right base image

A popular approach to this challenge is to have two types of base images: one used during development and unit testing and another for later stage testing and production. In later stage testing and production your image does not require build tools such as compilers (for example, Javac) or build systems (such as Maven) or debugging tools. In fact, in production, your image may not even require Bash.

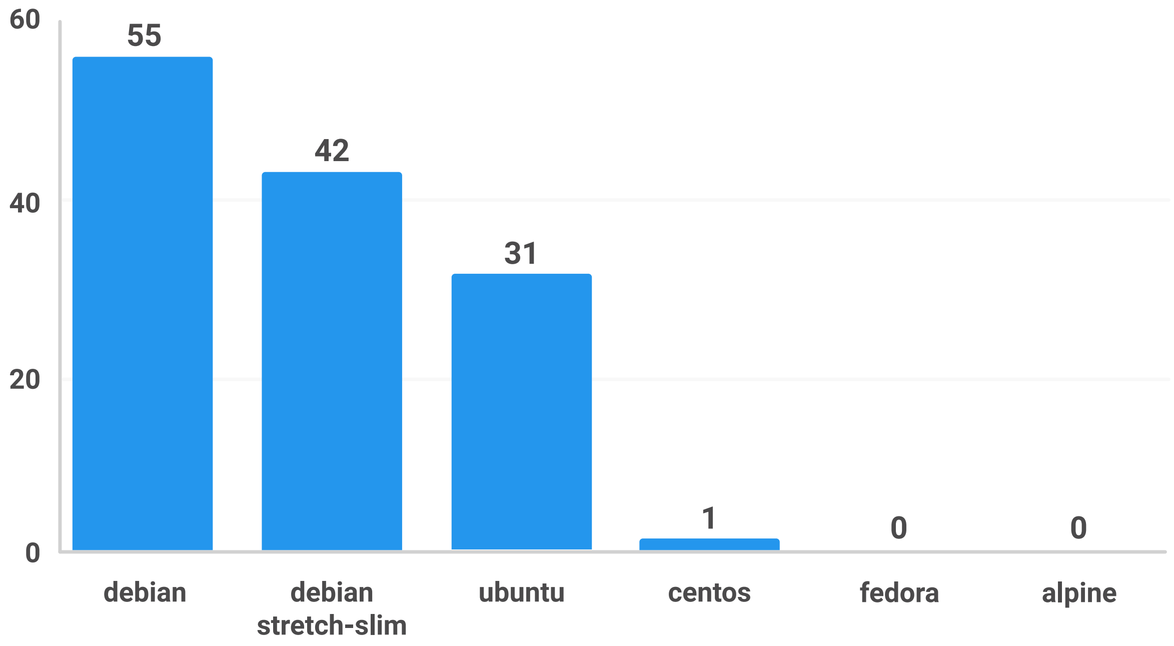
We see dramatic differences between the basic operating system images and the different variants. Most of the time, a full-blown operating system image is not necessary. Image build tools, like Buildah, allow you to build images from scratch and only install the packages you need and their dependencies. This lowers the attack plain on images considerably. Consider a Python application image with nothing but the Python package, it's dependent packages, and the Python application.

An additional advantage of Buildah is that it does not require the Docker daemon process. This is important in large scale container deployment platforms that reuse resources for building container images. The Docker daemon is a privileged process that has an open socket used to communicate with it. If the Docker daemon gets exploited then the node becomes compromised and often that means an entire cluster can be compromised. Either explicitly isolating build nodes or using a tool like Buildah will eliminate the need for a Docker daemon.

Selecting a stripped version or another implementation of a Linux distro can help trim down the number of vulnerabilities. When scanning the Alpine base image, a minimal 5MB in size Docker image based on Alpine Linux, we don't find any known vulnerability, however that is mostly due to the Alpine project not maintaining a security advisory program, and so if vulnerabilities are present there is no official advisory to share them on in any case. Still, Alpine makes a good case of a very minimal and stripped-down base image upon which to build.

While no vulnerabilities were detected in the version of the Alpine image we tested, that's not to say that it is necessarily free of security issues. Alpine Linux handles vulnerabilities differently than the other major distros, who prefer to backport sets of patches. At Alpine, they prefer rapid release cycles for their images, with each image release providing a system library upgrade.

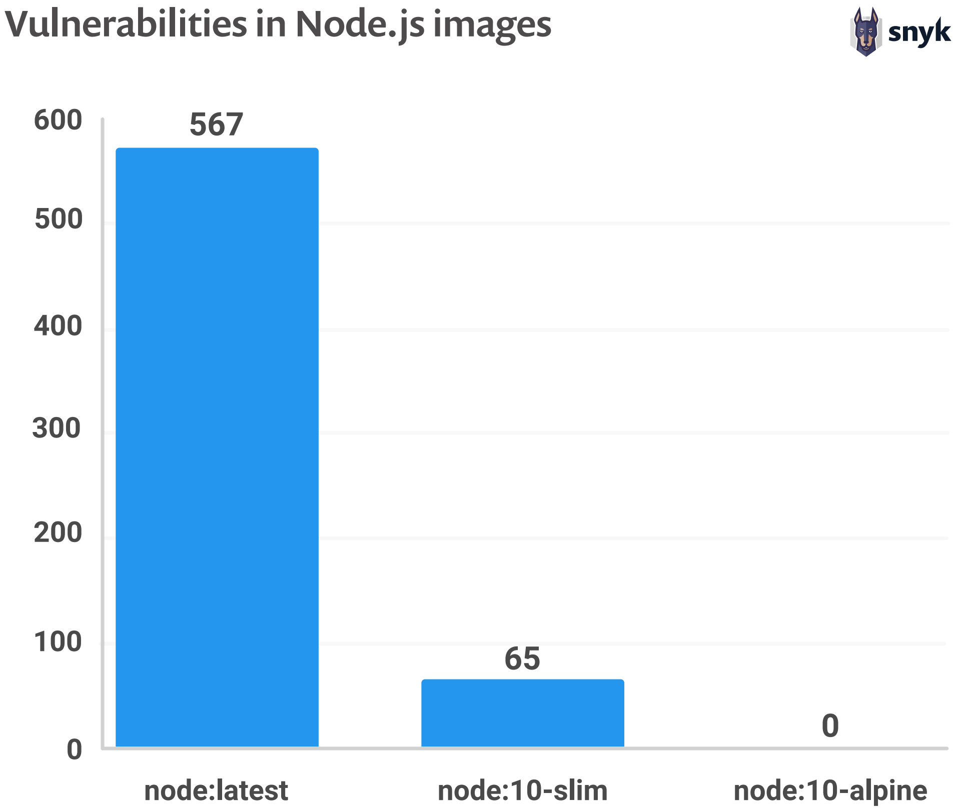
Vulnerabilities in OS images



As you can see in the graph above, changing the base image inside the Dockerfile or simply using another tag of a standard image can make a lot of difference.

When building your own image from a Dockerfile be sure that you do not depend on larger images than necessary. This shrinks the size of your image and also minimizes the number of vulnerabilities introduced through your dependencies.

Based on scans performed by Snyk users, we found that 44% of docker image scans had known vulnerabilities, for which there were newer and more secure base image available. This remediation advice is unique to Snyk. Developers can take action to upgrade their Docker images. Automating the process of scanning for newer or better base images and alerting to this can be considered a best practice.



Use multi-stage builds

Multi-stage builds are available when using Docker 17.05 and higher. These kinds of builds are designed to create an optimized Dockerfile that is easy to read and maintain.

With a multi-stage build, you can use multiple images and selectively copy only the artifacts needed from a particular image. You can use multiple FROM statements in your Dockerfile, and you can use a different base image for each FROM, copying the artifacts from one step to the next. You can leave the artifacts that you don't need behind and still end up with a concise final image.

This method of creating a tiny image does not only significantly reduce complexity but also the change of implementing vulnerable artifacts in your image. So instead of images that are built on images that again are built on other images, with multi-stage builds you are able to “cherry-pick” your artifacts without inheriting the vulnerabilities from the base images on which you rely. More information on how to build multi-stage builds can be found in the Docker docs.

As mentioned earlier in the report, another approach is to use tools like Buildah to create minimal production images with only the packages required to run your application. For more information on Buildah see Buildah.io and Podman and Buildah for Docker Users.

Key takeaways

With multi-stage builds, you only copy the artifacts you actually need

Rebuilding images

Every Docker image builds from a Dockerfile. These Dockerfiles for the Docker images on Docker Hub are publicly available on GitHub. A Dockerfile contains a set of instructions which allows you to automate the steps you would normally manually take to create an image. Additionally, some libraries may be imported and custom software can be installed. These are all instructions in the Dockerfile. In the State of Open Source Security 2019 report, we discovered that 20% of the Docker images with vulnerabilities could have been solved by a simple rebuild of the image.

Building your image is basically a snapshot of that image at that moment in time. When you depend on a base image without a strict tag, every time a rebuild is done the base image can be different. When packages are installed using a package installer, rebuilding can change the image.

A Dockerfile containing the following can potentially have a different binary with every rebuild.

```
FROM ubuntu:latest
RUN apt-get -y update && apt-get
install -y python
```

Key takeaways

20% of the vulnerable Docker images can be fixed by a rebuild

Any Docker image should be rebuilt regularly to prevent known vulnerabilities in your image that have already been solved. When rebuilding use the no-cache option `--no-cache` to avoid cache hits and to ensure a fresh download.

For example:

```
docker build --no-cache -t
myImage:myTag myPath/
```

In summary, follow these best practices when rebuilding your image:

- Each container should have only one responsibility.
- Containers should be immutable, lightweight and fast.
- Don’t store data in your container (use a shared data store).
- Containers should be easy to destroy and rebuild.
- Use a small base image (such as Linux Alpine).
- Smaller images are easier to distribute.
- Avoid installing unnecessary packages.
- This keeps the image nice, clean and safe.
- Avoid cache hits when building.
- Auto-scan your image before deploying with a tool like Snyk’s container scan to avoid pushing vulnerable containers to production.
- Scan your images daily both during development and production for vulnerabilities. Based on that, automate the rebuild of images if necessary.

Scanning images during development

Creating an image from a Dockerfile and even rebuilding an image can introduce new vulnerabilities in your system. Previously we saw that 68% of users believe that developers have a fair amount of responsibility in container security. Scanning your docker images during development should be part of your workflow to catch vulnerabilities as early as possible.

When thinking of shifting security left, developers should ideally be able to scan a Dockerfile and their images from their local machine before it is committed to a repository or a build pipeline.

This does not mean that you should replace CI-pipeline scans with local scans, but rather that it is preferable to scan at all stages of development, and preferably scans should be automated.
Think about automated scans during build, before pushing the image to a registry and pushing an image to a production environment. Refusing that an image go into a registry or enter the production system because the automated scan found new vulnerabilities should be considered a best-practice.

Snyk's recently released container vulnerability management scan Docker images by extracting the image layers and inspecting the package manager manifest info. We then compare every OS package installed in the image against our Docker vulnerability database. In addition to that, it is also crucial to scan the key binaries installed on the images. Snyk also supports scanning the key binaries that are often not installed by the OS package manager (dpkg, RPM and APK), but by other methods such as a RUN command.

When giving developers the tools to scan Dockerfiles and images during development on their local machines, another layer of protection is created and developers can actively contribute to a more secure system in general.

Scanning containers during production

It turns out that 91% of respondents do not scan their docker images in production. Actively checking your container could save you a lot of trouble when a new vulnerability is discovered and your production system might be at risk.

Periodically (for example, daily) scanning your docker image is possible by using the Snyk monitor capabilities for containers. Snyk creates a snapshot of the image's dependencies for continuous monitoring.

Additionally you should also activate runtime monitoring. Scanning for unused modules and packages inside your runtime gives insight about how to shrink images. Removing unused components prevents unnecessary vulnerabilities from entering both system and application libraries. This also makes an image more easily maintainable.

Snyk helps you use open source and stay secure.

Get started at snyk.io

Twitter: @snyksec

Web: https://snyk.io

Office info

London

1 Mark Square
London EC2A 4EG

Tel Aviv

40 Yavne st.,
first floor

Boston

WeWork 9th Floor
501 Boylston St
Boston, MA 02116

Report authors

Brian Vermeer (@BrianVerm), Snyk

William Henry (@ipbabble), RedHat

Report design

Growth Labs (@GrowthLabsMKTG)