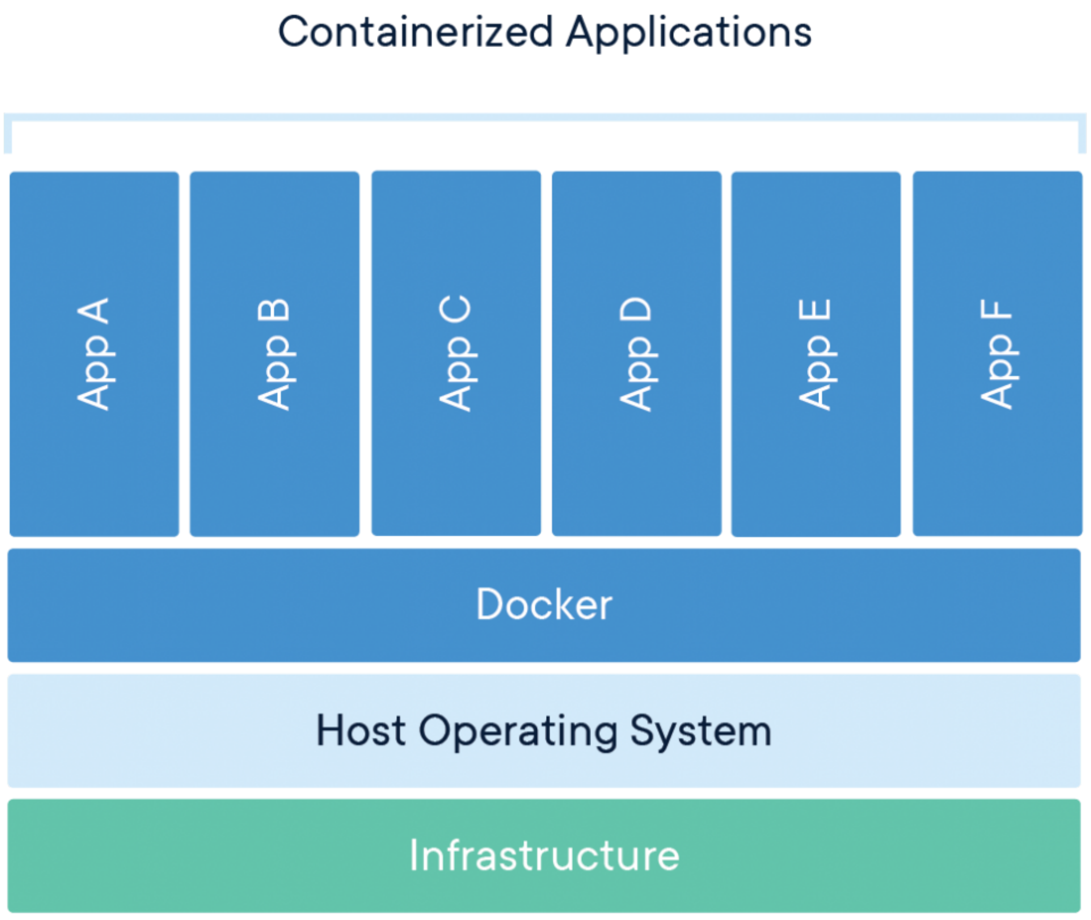
Docker and Kubernetes — root vs. privileged

Bryant Hagadorn Follow
Jun 26 · 4 min read ★

Most of us that are familiar with a Unix system, like macOS or Linux, are used to casually elevating our privileges to the `root` user through the use of `sudo` . Usually this happens when debugging development tools or trying to edit files in directories that are protected. It's almost default behavior to try an unsuccessful command as `sudo` after the first attempt for many (including me!).



Fundamental to understanding Docker security is understanding what containers actually are (hint: not VM's!)

Docker offers a seemingly similar `--privileged` flag, which is actually *much* different from casual `sudo` usage, that might expose your applications to unnecessary risk. I'm going to show you how this is much different than running as `root` (and how to avoid running as root!) as well as what privileged actually means.

Running as Root

Docker allows isolation of a process, capabilities, and filesystem on its host OS, and for practical reasons most containers actually run as `root` by default. Just to sample, I'll take the three most popular images on DockerHub at the time of this writing in case you are curious, looking for the named user as well as the User ID:

Postgres:

```
$ docker run -it postgres
# whoami
root
# id -u
0
```

Couchbase:

```
$ docker run -it couchbase sh
# whoami
root
# id -u
0
```

Alpine:

```
$ docker run -it alpine sh
# whoami
root
# id -u
0
```

As you can see, most images run as root by default. This typically allows easier debugging especially if you are going to `exec` into the containers. Whilst the `root` user is shipped with very limited Linux capabilities, it is best to avoid running as root.

Avoiding Running as Root

Whilst running as `root` inside of the container is actually quite normal, it should still be avoided if you're trying to harden your containers. First, it violates the principles of least privilege. Second, and more technically, the container will be a part of the same user namespace that ran the Docker command, and if the container is able to escape, it will have access to the same resources like volumes and sockets.

There are two ways to avoid running as root:

by tweaking the Dockerfile to use a specific user:

```
// Dockerfile

FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

or overriding the User ID at runtime:

```
$ docker run -it --user 4000 postgres sh
# whoami
whoami: cannot find name for user ID 4000
# id -u
4000
```

How about Privileged?

The `--privileged` flag maps the User ID we saw earlier of 0 directly to the User ID 0 of the host and gives it unfettered access to any system call of its choosing. You see, in normal operation, even with `root` inside the container, Docker restricts the container's Linux capabilities. This restriction includes things like **CAP_AUDIT_WRITE**, for example, which allows overwriting the Kernel's audit logs — power that is highly unlikely needed by your containerized workloads. Really, privileged should only be used in specific settings where you really need it, and in short it gives the container access *to almost everything the host (as root) can do*. Essentially it's a free pass to escape the container's contained file system, processes, sockets and otherwise contained items. It has specific use cases such as Docker in Docker, other CI/CD tooling requirements (where the Docker daemon is needed from inside of a Docker container), and places where extreme networking is needed. Even so, there are ways to avoid it — GitLab, for example, suggests an alternative to privileged pods called Kaniko created by Google Container Tools. As well, the NestyBox product gives users a secure and efficient Docker in Docker experience.

Lets see an example using the Ubuntu image (inside of a VM so we can't break anything):

Without privileged:

```
$ docker run -it ubuntu sh
# whoami
root # Notice here, we are still root!
# id -u
0
# hostname
382f1c400bd
# sysctl kernel.hostname=Attacker
sysctl: setting key "kernel.hostname": Read-only file system  # Yet
we can't do this
```

With privileged:

```
$ docker run -it --privileged ubuntu sh
# whoami
root. # Root again
# id -u
0
# hostname
86c62e9bba5e
# sysctl kernel.hostname=Attacker
kernel.hostname = Attacker # Except now we are privileged
# hostname
Attacker
```

Kubernetes offers the same functionality via the Security Context :

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    securityContext:
      privileged: true
```

As well, Kubernetes contains an enforcement mechanism called PodSecurityPolicies which are an admission controller (aka something Kubernetes checks before allowing the container into the cluster). One highly recommended policy would be not allowing privileged Pods:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false  # Don't allow privileged pods!
```

Summary

Hopefully by the end of this you learned a little bit more about `root` and the `--privileged` flag, as well as their relation to the "host" OS, and whether you are trying to clamp down on the security of your containers or debug an issue, you know enough to keep your applications safe. Defensive security is all about defense in depth (layers, like an onion) and reducing your attack surface — by not running as root, not running as privileged, and adding SecurityContext and PodSecurityPolicies are four great layers to achieving greater container security. **Please be sure to follow me if you like reading about this type of stuff** and shoot me an email at bryant.hagadorn@gmail.com if you'd like to ask any questions, comment, or anything else. Thanks!

Kubernetes Docker Security Linux Containers

App Store Google Play