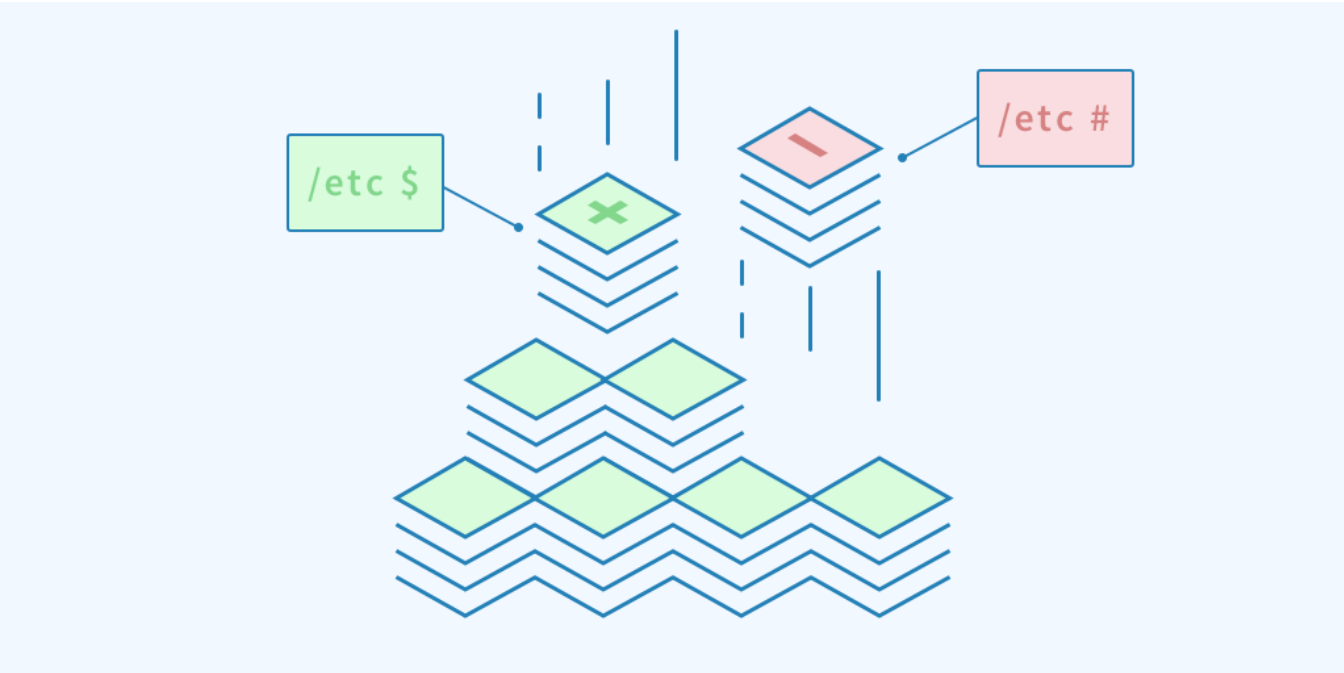
Marc Campbell

Follow 618 Followers About

Processes In Containers Should Not Run As Root

Marc Campbell Sep 28, 2017 · 5 min read



tldr;

Processes in a container should not run as root, or assume that they are root. Instead, create a user in your Dockerfile with a known UID and GID, and run your process as this user. Images that follow this pattern are easier to run securely by limiting access to resources.

Overview

Most containerized processes are application services and therefore don't require root access. While Docker requires root to run, containers themselves do not. Well written, secure and reusable Docker images should not expect to be run as root and should provide a predictable and easy method to limit access.

Why This Matters

Remember that a process running in a container is no different from other process running on Linux, except it has a small piece of metadata that declares that it's in a container. Containers are not trust boundaries, so therefore, anything running in a container should be treated with the same consideration as anything running on the host itself.

Just like you wouldn't (or shouldn't) run anything as root on your server, you shouldn't run anything as root in a container on your server. Running binaries that were created elsewhere requires a significant amount of trust, and the same is true for binaries in containers.

If a process inside a container runs as root by default it's possible to change the uid and gid when starting the container. As the author of the image, you should default to running as a different user and make it easier to limit access for that user. By creating a user in your Dockerfile, you are making it not only secure by default but also easier to keep secure.

An example will show the risk of running a container as root. Let's create a file in the `/root` directory, preventing anyone other than `root` 1from viewing it:

```
marc@srv:~$ sudo -s

root@srv:~# cd /root
root@srv:~# echo "top secret stuff" >> ./secrets.txt
root@srv:~# chmod 0600 secrets.txt
root@srv:/root# ls -l
total 4
-rw------- 1 root root 17 Sep 26 20:29 secrets.txt

root@srv:/root# exit
exit

marc@srv:~$ cat /root/secrets.txt
cat: /root/secrets.txt: Permission denied
```

I now have a file named `/root/secrets.txt` that only root can see. I'm logged in as a normal (non-root) user. Let's create a Docker image from this Dockerfile:

```
FROM debian:stretch
CMD ["cat", "/tmp/secrets.txt"]
```

And finally, let's run this Dockerfile, bind-mounting a volume from the `/root/secrets.txt` file that I cannot read to the `/tmp/secrets.txt` file inside the container:

```
marc@srv:~$ docker run -v /root/secrets.txt:/tmp/secrets.txt <img>
top secret stuff
```

Even though I'm `marc`, the container is running as `root` and therefore has access to everything `root` has access to on this server. This isn't ideal; running containers this way means that every container you pull from Docker Hub could have full access to everything on your server (depending on how you run it).

Recommendation

The recommendation here is to create a user with a known `uid` in the Dockerfile and run the application process as that user. The start of a Dockerfile should follow this pattern:

```
FROM <base image>

RUN groupadd -g 999 appuser && \
    useradd -r -u 999 -g appuser appuser
USER appuser

... <rest of Dockerfile> ...
```

Using this pattern, it's easy to run a container in the context of a user/group with the least privileges required.

For example, I'll add that to my Dockerfile from above and re-run the example. My Dockerfile now looks like this:

```
FROM debian:stretch

RUN groupadd -g 999 appuser && \
    useradd -r -u 999 -g appuser appuser
USER appuser

CMD ["cat", "/tmp/secrets.txt"]
```

Running this container with the same command as before:

```
marc@srv:~$ docker run -v /root/secrets.txt:/tmp/secrets.txt <img>
cat: /tmp/secrets.txt: Permission denied
```

Now, the default behavior of this container is that it does not have `root` privileges on the host.

Reusing Other Images

Docker images are great because they are reusable. But when you `FROM` an image that is running as non-root, your container will inherit that non-root user. If you need to create your own or perform operations as root, be sure to `USER root` somewhere near the top of your Dockerfile. Then `FROM appuser` again to make it usable.

Running Other Containers As Non-Root Users

Docker images are designed to be portable, and it's normal to pull other images from Docker Hub to use. Some of these (official images) will follow this best practices and run as a normal user account. But many images don't do this. Many images just run as `root` and leave it up to you to figure out how to securely run them. There are a couple of options that will allow you to securely run an image that doesn't create it's own user.

Create Another Image

First, one option is to create another image, using the original image as the `FROM` layer. You can then create a user account, and copy the original `ENTRYPOINT` and `CMD` directives to your own image. This resulting image now follows the best practice outlined here, and will run securely by default. The tradeoff here is that you need to rebuild your image when the base image is updated. You will have to set up a process to rebuild when the base image is rebuilt.

Specify a `uid` When Starting The Container

Finally, you can create a user on the host, and pass its `uid` to Docker when starting the container. For example, revisiting the original example Dockerfile:

```
FROM debian:stretch
CMD ["cat", "/tmp/secrets.txt"]
```

I can run this container with or without a user id parameter and see the different results (user id 1001 is my own account on this server):

```
$ docker run ––user 1001 –v /root/secrets.txt:/tmp/secrets.txt <img>
cat: /tmp/secrets.txt: Permission denied

$ docker run –v /root/secrets.txt:/tmp/secrets.txt <img>
top secret stuff
```

This works and does the same thing as creating a user in the Dockerfile, but it requires the user to optionally run the container securely. Specifying a non-root user in the Dockerfile will make the container run securely *by default*.

Further Reading

Understanding how uid and gid work in Docker containers

Official reference to the Docker Run command (Docker)

Docker Security

—-

If you're using Docker to ship your SaaS application, you should check out www.replicated.com to power an enterprise, installable version of your product.

Docker DevOps Containers Security