Docker Security: Vulnerabilities and Best Practices

A guide to follow best practices while working with Docker

We'll cover the following

Docker has provided tremendous benefits over the traditional hypervisors. However, most of its components are shared with the host kernel. So, if proper security measures are not taken, the host system can be at risk of compromisation and let an attacker take control of it.

In this lesson, we will see some Docker vulnerabilities and best practices along with some security tips.

Risks

There are multiple risks associated with Docker containers and images.

Host compromisation

Since containers use the host's kernel as a shared kernel for running processes, a

compromised container kernel can exploit or attack the entire host system.

Container breakouts

If somehow, a user is able to escape the container namespace, it will be able to interact with the other processes on the host and can stop or kill the processes.

Max resource utilization

Sometimes a container uses all the resources of the host machine if it is not restricted. This will force other services to halt and stop the execution.

Attack using untrusted images

Docker allows us to run all images present on Docker Hub as well a local build. So, when an image from an untrusted source is run on the machine, the attacker's malicious program may get access to the kernel or steal all the data present in container and mounted volumes.

Best practices to mitigate risks

No matter what measures you take, security can be breached at any level and no one can totally remove the security risks. However, we can mitigate the risks by following some best processes to ensure that we close all the gates for an attacker to get access to our host machine.

Container lifecycle management

Through the container lifecycle management process, we establish a strong foundation for the review process of creating, updating, and deleting a container. This takes care of all the security measures at the start while creating a container. When a container is updated instead of reviewing only the updated layer, we should review all layers again.

Information management

Never push any sensitive data like passwords, ssh keys, tokens, certificates in an image. It should be encrypted and kept inside a secret manager. Access to these secrets should be explicitly provided to the services and only when they are running.

No root access

A container should never be run with root-level access. A role-based access control system will reduce the possibility of accidental access to other processes running in the same namespace. Many of the organizations restrict access using the active

directory and provide appropriate access based on user roles.

In general, we can use Linux's inbuilt commands to create a temporary non-root user on the fly.

```
FROM python:3.5
RUN groupadd -r myuser && useradd -r -g myuser myuser
<HERE DO WHAT YOU HAVE TO DO AS A ROOT USER LIKE INSTALLING PACKAGES ETC.>
USER myuser
```

or while running a container from the image use,

```
docker run -u 4000 python:3.5
```

This will run the container as a non-root user.

Trusted image source

Check the authenticity of every image pulled from Docker Hub. Use Docker Content Trust to check the authenticity of the Image. Docker Content Trust is a new feature incorporated into Docker 1.8. It is disabled by default, but once enabled, allows you to verify the integrity, authenticity, and publication date of all Docker images from the Docker Hub registry.

Enable Docker content trust using `export DOCKER_CONTENT_TRUST=1` and try to pull this unsigned image `docker pull jpetazzo/clock`.

```
# docker pull jpetazzo/clock
Using default tag: latest
Error: remote trust data does not exist for docker.io/jpetazzo/clock: notary.d
ocker.io does not have trust data for docker.io/jpetazzo/clock
```

Docker will check whether the image is safe or not and will throw an error if it is not.

Limit resource usage

One of the best features I like in Docker is the flexibility to assign a predefined resource usage policy for a container. This is supported only in swarm mode. Do you remember the `deploy` section of our docker-compose.yml file?

We can limit the CPU usage of a container by declaring the allowed usage in the deploy section. Here is an example from the official documentation

deploy section. Here is an example from the official documentation.

```
version: "3.8"
services:
  redis:
    image: redis:alpine
    deploy:
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
```

This way, we will be sure that no container will use host resources above a threshold and other services will be running on the host without any harm.

Cautionary use of Docker socket

In the visualizer service, we have mounted Docker socket `/var/run/docker.sock` on the container. Bind mounting the Docker daemon socket gives a lot of power to a container as it can control the daemon. It must be used with caution and only with containers we can trust. There are a lot of third-party tools that demand this socket to be mounted while using their service.

You should verify such services with Docker Content Trust and vulnerability management processes before using them.

We mounted the Docker socket on the visualizer service container as the visualizer image is Docker's official image and it can be verified by Docker Content Trust and Docker bench for Security.

Mounting volumes as read only

One of the best practices is to mount the host filesystem as read-only if there is no data being saved by the container. We can do that by simply using a `ro` flag for mounts using `-v` argument or `readonly` with the `--mount` argument.

Example:

```
$ docker run -v volume-name:/path/in/container:ro python:3.5
```

or

```
$ docker run --mount source=volume-name,destination=/path/in/container,readonl
y python:3.5
```

In the next lesson, you will see this in action when we will deploy our Flask app as a production-ready app.

The best practices are implemented by organizations as a part of their standard development process and developers need to follow the rules. Using these practices as an independent developer will also improve the code quality. In the next lesson, we will deploy our app after making some changes to it which needs to be done for a production-ready app. So, when you absorb what you learnt in this lesson, hop on to the next and final lesson to check the production deployment.