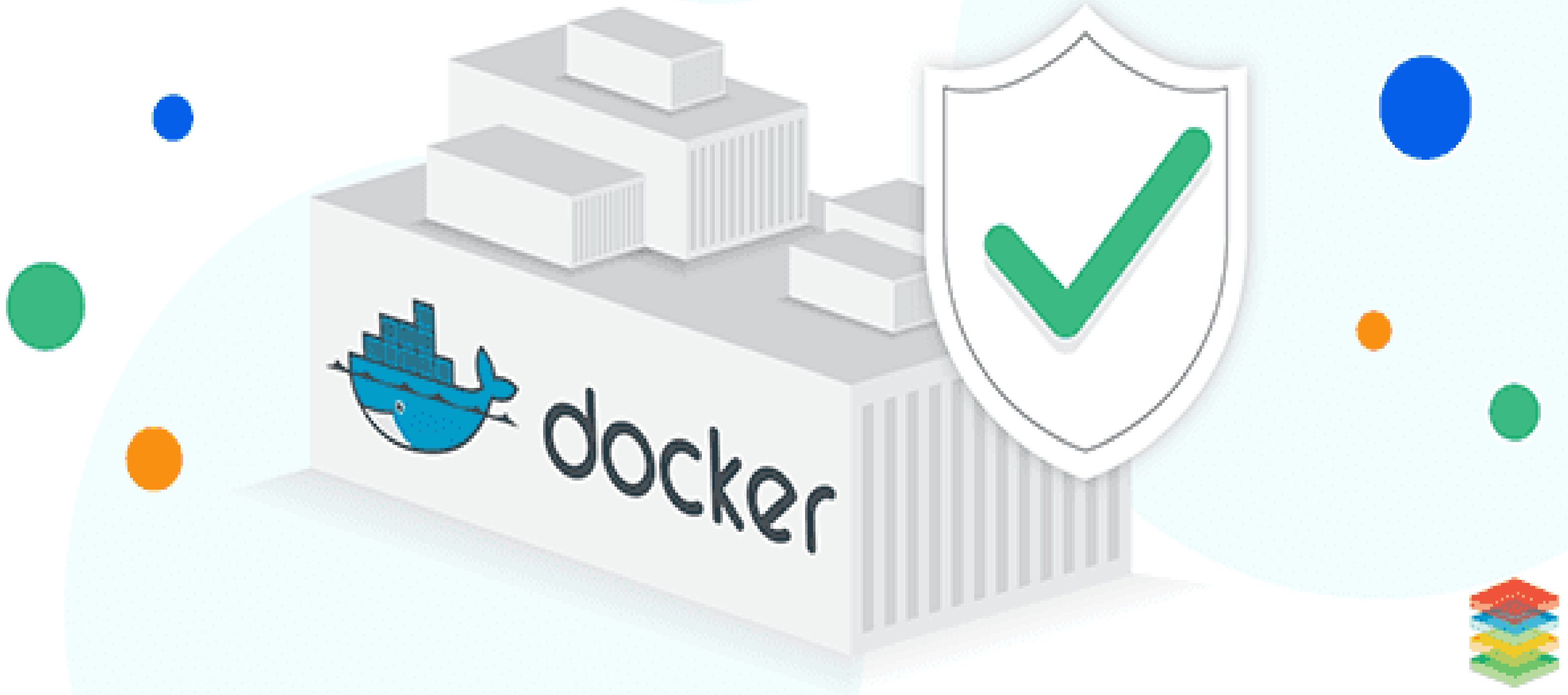
Container Security



The challenge of Docker container security

Before Docker, most organizations used virtual machines or bare-metal servers to host applications. From a security perspective, these technologies are relatively simple. You need to focus on just two layers (the host environment and the application) when hardening your deployment and monitoring for security-relevant events.

You also typically do not need to worry much about APIs, overlay networks or complex software-defined storage configurations, because these are not usually a major part of virtual-machine or bare-metal deployments.

Docker Container Security

Docker container security is more complicated, largely because a typical Docker environment has many more moving parts. Those parts include:

- Your containers. You probably have multiple Docker container images, each hosting individual microservices. You probably also have multiple instances of each image running at a given time. Each of those images and instances needs to be secured and monitored separately.
- The Docker daemon, which needs to be secured to keep the containers it hosts safe.
- The host server, which could be bare metal or a virtual machine.
- If you host your containers in the cloud using a service like ECS, that is another layer to secure.
- Overlay networks and APIs that facilitate communication between containers.
- Data volumes or other storage systems that exist externally from your containers.

So, Docker security truly is more complicated than other security strategies.

Best practices

#1 Set resource quotas

- One handy thing that Docker makes easy to do is to configure resource quotas on a per-container basis. Resource quotas allow you to limit the amount of memory and CPU resources that a container can consume.
- This feature is useful for several reasons. It can help to keep your Docker environment efficient and prevent one container or application from hogging system resources. But it also enhances security by preventing a compromised container from consuming a large amount of resources in order to disrupt service or perform malicious activities.

#2 Don’t run as root

Use sudo user with certain permission

Best practices

#3 Secure your container registries

Container registries are part of the reason Docker is so powerful. use role-based access control to define explicitly who can access what, and blacklist access from everyone else.

#4 Use trusted, secure images

ou should also be sure that the container images you pull come from a trusted source. You can also take advantage of image scanning tools to help identify some known vulnerabilities within Docker images. Most enterprise-level container registries have built-in scanning tools. Some of them, like Clair, can be used separately from a registry to scan individual images, too.

#5 Identify the source of your code

Best practices

#6 API and network security

Docker containers typically rely heavily on APIs and networks to communicate with each other. That’s why it’s essential to make sure that your APIs and network architectures are designed securely, and that you monitor the APIs and network activity for anomalies that could indicate an intrusion.

Resource Quotas

- When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.
- Resource quotas are a tool for administrators to address this concern.
- A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that project.

Watchtower

- Watchtower is an application that will monitor your running Docker containers and watch for changes to the images that those containers were originally started from. If watchtower detects that an image has changed, it will automatically restart the container using the new image.

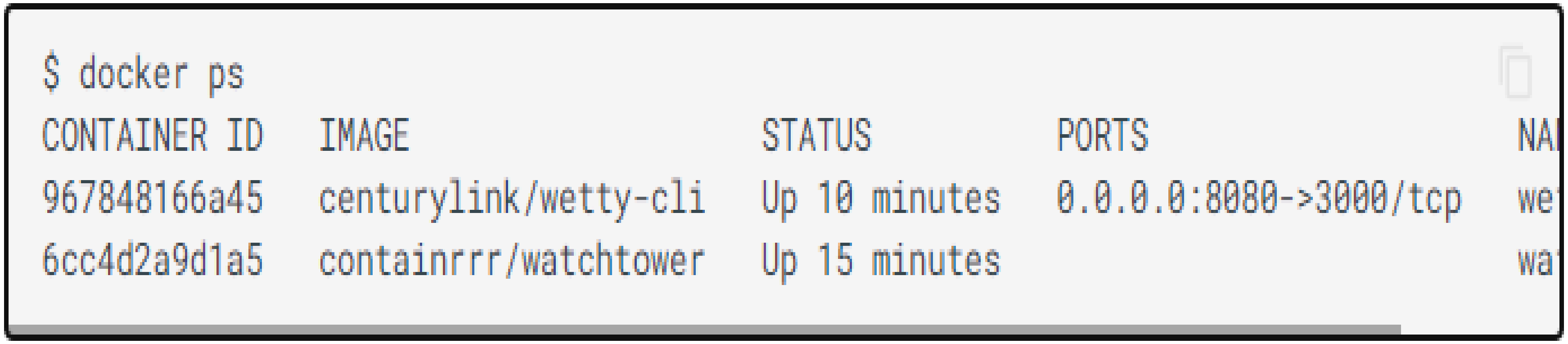
- With watchtower you can update the running version of your containerized app simply by pushing a new image to the Docker Hub or your own image registry. Watchtower will pull down your new image, gracefully shut down your existing container and restart it with the same options that were used when it was deployed initially.

Usage overview

- Watchtower is itself packaged as a Docker container so installation is as simple as pulling the containrrr/watchtower image.
- Since the watchtower code needs to interact with the Docker API in order to monitor the running containers, you need to mount */var/run/docker.sock* into the container with the -v flag when you run it.
- Run the watchtower container with the following command:

```
docker run -d  \
--name watchtower \
-v /var/run/docker.sock:/var/run/docker.sock \
containrrr/watchtower
```

Usage overview


```
$ docker ps
CONTAINER ID   IMAGE                   STATUS           PORTS                    NA
967848166a45   centurylink/wetty-cli   Up 10 minutes    0.0.0.0:8080->3000/tcp   we
6cc4d2a9d1a5   containrrr/watchtower   Up 15 minutes                             wa
```

Every few minutes watchtower will pull the latest *centurylink/wetty-cli* image and compare it to the one that was used to run the "wetty" container. If it sees that the image has changed it will stop/remove the "wetty" container and then restart it using the new image and the same docker run options that were used to start the container initially (in this case, that would include the -p 8080:3000 port mapping).

Arguments

By default, watchtower will monitor all containers running within the Docker daemon to which it is pointed. However, you can restrict watchtower to monitoring a subset of the running containers by specifying the container names as arguments when launching watchtower.

```
$ docker run -d \
 --name watchtower \
-v /var/run/docker.sock:/var/run/docker.sock \
containrrr/watchtower \
nginx redis
```

Portainer – Docker GUI

INTRODUCTION TO PORTAINER

Build, manage and support your containerized environments with Portainer

- ***Portainer*** is a lightweight management UI which allows you to **easily** manage your Docker host or Swarm cluster.
- ***Portainer*** is meant to be as **simple** to deploy as it is to use. It consists of a single container that can run on any Docker engine (Docker for Linux and Docker for Windows are supported).
- ***Portainer*** allows you to manage your Docker stacks, containers, images, volumes, networks and more ! It is compatible with the *standalone Docker* engine and with *Docker Swarm*.

Portainer – Docker GUI

Open dockerhub and check portainer/portainer

```
# docker pull portainer/portainer
```

```
# docker run -d -p 9000:9000 -p 8000:8000 --name portainer --restart always -v
/var/run/docker.sock:/var/run/docker.sock portainer/portainer
```

Now Copy Instance Public IP –Paste in URL -- IP/9000 --Enter