Docker Security Cheat Sheet

Introduction

Docker is the most popular containerization technology. Upon proper use, it can increase the level of security (in comparison to running applications directly on the host). On the other hand, some misconfigurations can lead to downgrade the level of security or even introduce new vulnerabilities.

The aim of this cheat sheet is to provide an easy to use list of common security mistakes and good practices that will help you secure your Docker containers.

Rules

RULE #0 - Keep Host and Docker up to date

To prevent from known, container escapes vulnerabilities, which typically end in escalating to root/administrator privileges, patching Docker Engine and Docker Machine is crucial.

In addition, containers (unlike in virtual machines) share the kernel with the host, therefore kernel exploits executed inside the container will directly hit host kernel. For example, kernel privilege escalation exploit (like Dirty COW) executed inside a well-insulated container will result in root access in a host.

RULE #1 - Do not expose the Docker daemon socket (even to the containers)

Docker socket */var/run/docker.sock* is the UNIX socket that Docker is listening to. This is the primary entry point for the Docker API. The owner of this socket is root. Giving someone access to it is equivalent to giving unrestricted root access to your host.

Do not enable *tcp* Docker daemon socket. If you are running docker daemon with `-H tcp://0.0.0.0:XXX` or similar you are exposing un-encrypted and unauthenticated direct access to the Docker daemon, if the host is internet connected this means the docker daemon on your computer can be used by anyone from the public internet. If you really, **really** have to do this, you should secure it. Check how to do this following Docker official documentation.

Do not expose */var/run/docker.sock* to other containers. If you are running your docker image with `-v /var/run/docker.sock://var/run/docker.sock` or similar, you should change it. Remember

that mounting the socket read-only is not a solution but only makes it harder to exploit. Equivalent in the docker-compose file is something like this:

```
volumes:
- "/var/run/docker.sock:/var/run/docker.sock"
```

RULE #2 - Set a user

Configuring the container to use an unprivileged user is the best way to prevent privilege escalation attacks. This can be accomplished in three different ways as follows:

1. During runtime using `-u` option of `docker run` command e.g.:

```
docker run -u 4000 alpine
```

2. During build time. Simple add user in Dockerfile and use it. For example:

```
FROM alpine
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
```

3. Enable user namespace support (`--userns-remap=default`) in Docker daemon

More information about this topic can be found at Docker official documentation

In kubernetes, this can be configured in Security Context using `runAsNonRoot` field e.g.:

```
kind: ...
apiVersion: ...
metadata:
  name: ...
spec:
  ...
  containers:
  - name: ...
    image: ....
    securityContext:
          ...
          runAsNonRoot: true
          ...
```

As a Kubernetes cluster administrator, you can configure it using Pod Security Policies.

RULE #3 - Limit capabilities (Grant only specific capabilities, needed by a container)

Linux kernel capabilities are a set of privileges that can be used by privileged. Docker, by default, runs with only a subset of capabilities. You can change it and drop some capabilities (using `--cap-drop`) to harden your docker containers, or add some capabilities (using `--cap-add`) if needed. Remember not to run containers with the `--privileged` flag - this will add ALL Linux kernel capabilities to the container.

The most secure setup is to drop all capabilities `--cap-drop all` and then add only required ones. For example:

```
docker run --cap-drop all --cap-add CHOWN alpine
```

And remember: Do not run containers with the *--privileged* flag!!!

In kubernetes this can be configured in Security Context using `capabilities` field e.g.:

```
kind: ...
apiVersion: ...
metadata:
  name: ...
spec:
  ...
  containers:
  - name: ...
    image: ....
    securityContext:
          ...
          capabilities:
            drop:
              - all
            add:
              - CHOWN
          ...
```

As a Kubernetes cluster administrator, you can configure it using Pod Security Policies.

RULE #4 - Add –no-new-privileges flag

Always run your docker images with `--security-opt=no-new-privileges` in order to prevent escalate privileges using `setuid` or `setgid` binaries.

In kubernetes, this can be configured in Security Context using `allowPrivilegeEscalation` field e.g.:

```
kind: ...
apiVersion: ...
metadata:
  name: ...
spec:
  ...
  containers:
  - name: ...
    image: ....
    securityContext:
          ...
          allowPrivilegeEscalation: false
          ...
```

As a Kubernetes cluster administrator, you can refer to Kubernetes documentation to configure it using Pod Security Policies.

RULE #5 - Disable inter-container communication (–icc=false)

By default inter-container communication (icc) is enabled - it means that all containers can talk with each other (using `docker0` bridged network). This can be disabled by running docker daemon with `--icc=false` flag. If icc is disabled (icc=false) it is required to tell which containers can communicate using --link=CONTAINER_NAME_or_ID:ALIAS option. See more in Docker documentation - container communication

In Kubernetes Network Policies can be used for it.

RULE #6 - Use Linux Security Module (seccomp, AppArmor, or SELinux)

First of all, do not disable default security profile!

Consider using security profile like seccomp or AppArmor.

Instructions how to do this inside Kubernetes can be found at Security Context documentation and in Kubernetes API documentation

RULE #7 - Limit resources (memory, CPU, file descriptors, processes, restarts)

The best way to avoid DoS attacks is by limiting resources. You can limit memory, CPU, maximum number of restarts (`--restart=on-failure:<number_of_restarts>`), maximum number of file descriptors (`--ulimit nofile=<number>`) and maximum number of processes (`--ulimit nproc=<number>`).

Check documentation for more details about ulimits

You can also do this inside Kubernetes: Assign Memory Resources to Containers and Pods, Assign CPU Resources to Containers and Pods and Assign Extended Resources to a Container

RULE #8 - Set filesystem and volumes to read-only

Run containers with a read-only filesystem using `--read-only` flag. For example:

```
docker run --read-only alpine sh -c 'echo "whatever" > /tmp'
```

If an application inside a container has to save something temporarily, combine `--read-only` flag with `--tmpfs` like this:

```
docker run --read-only --tmpfs /tmp alpine sh -c 'echo "whatever" > /tmp/file'
```

Equivalent in the docker-compose file will be:

```
version: "3"
services:
  alpine:
    image: alpine
    read_only: true
```

Equivalent in kubernetes in Security Context will be:

```
kind: ...
apiVersion: ...
metadata:
  name: ...
spec:
  ...
  containers:
  - name: ...
    image: ....
    securityContext:
          ...
          readOnlyRootFilesystem: true
          ...
```

In addition, if the volume is mounted only for reading **mount them as a read-only** It can be done by appending `:ro` to the `-v` like this:

```
docker run -v volume-name:/path/in/container:ro alpine
```

Or by using `--mount` option:

```
docker run --mount source=volume-name,destination=/path/in/container,readonly
alpine
```

RULE #9 - Use static analysis tools

To detect containers with known vulnerabilities - scan images using static analysis tools.

- Free
  - Clair
  - ThreatMapper
  - Trivy
- Commercial
  - Snyk **(open source and free option available)**
  - anchore **(open source and free option available)**
  - Aqua Security's MicroScanner **(free option available for rate-limited number of scans)**
  - JFrog XRay
  - Qualys

To detect secrets in images:

- ggshield **(open source and free option available)**
- SecretScanner **(open source)**

To detect misconfigurations in Kubernetes:

- kubeaudit
- kubesec.io
- kube-bench

To detect misconfigurations in Docker:

- inspec.io
- dev-sec.io

RULE #10 - Set the logging level to at least INFO

By default, the Docker daemon is configured to have a base logging level of 'info', and if this is not the case: set the Docker daemon log level to 'info'. Rationale: Setting up an appropriate log level, configures the Docker daemon to log events that you would want to review later. A base log level of 'info' and above would capture all logs except the debug logs. Until and unless required, you should not run docker daemon at the 'debug' log level.

To configure the log level in docker-compose:

```
docker-compose --log-level info up
```

Rule #11 - Lint the Dockerfile at build time

Many issues can be prevented by following some best practices when writing the Dockerfile. Adding a security linter as a step in the the build pipeline can go a long way in avoiding further headaches. Some issues that are worth checking are:

- Ensure a `USER` directive is specified
- Ensure the base image version is pinned
- Ensure the OS packages versions are pinned
- Avoid the use of `ADD` in favor of `COPY`
- Avoid curl bashing in `RUN` directives

References:

- Docker Baselines on DevSec
- Use the Docker command line
- Overview of docker-compose CLI
- Configuring Logging Drivers
- View logs for a container or service
- Dockerfile Security Best Practices

Related Projects

OWASP Docker Top 10