Container Security: What Could Possibly Go Wrong?

Michaela Doležalová
Daniel Kouřil

Masaryk University, CESNET
What is a container?

- fundamentally, a container is just a running process
- it is isolated from the host and from other containers
- each container usually interacts with its own private filesystem
- there are different containerization technologies available (Docker, LXD, Podman, Singularity, ...)
- in this tutorial, we will focus mainly on Docker
Containers vs. Virtual Machines

- A container is an abstraction of the application layer (it runs natively on Linux)

- A virtual machine is an abstraction of the hardware layer (it runs a full-blown “guest” operating system)
Threat Landscape

- proper **deployment** and **configuration** requires understanding the technology
- **image management** (integrity and authenticity of the image)
- trust in the **image maintainer** and the **repository operator**
- **malicious images** may be found even in an official registry

[Image Link]

https://unit42.paloaltonetworks.com/cryptojacking-docker-images-for-mining-monero/
Usual Best Practice

- especially proper **vulnerability/patch management**
- it is often kernel-related and therefore requiring reboot
- updates **not always** available
- **extremely important** (couple of vulns over the past few years)
- out of scope for today

Let’s move to Docker itself....
Docker Terminology

- **Docker container image** - a lightweight, standalone, executable package of software that includes everything needed to run an application (code, runtime, system tools, system libraries and settings)

- an image is usually pulled from a **registry** to a host machine (e.g. **DockerHub** - something like a Google Play store, Apple store, etc.)

- **Docker container** - an instance of an image

- a host machine runs the **container engine** (**Docker Daemon**)
Docker Architecture
Docker Container Creation

- the image is opened up and the **filesystem** of that image is copied into a **temporary archive** on the host
- Docker filesystem is a **stacked file system** of individual layers stacked on “mount”
- the ‘/’ root directory of the container is **mounted and available** on the host

```
/var/lib/docker/overlay2/51415bc9cd3ab2c47d218a897516ea2bf0545595fadf4a167ed5cf3230a5f99/
```

- changes to the directory **are visible** from both sides
- when the container is removed, any changes to its state **disappear** unless “committed” via **dockerd**
Docker Container Processes

- the container engine manages the process tree **natively** on the kernel
- to provide application sandboxing, Docker uses Linux **namespaces** and **cgroups**
- when you start a container with `docker run`, Docker creates a **set of namespaces** and **control groups**
Namespaces

- Docker Engine uses the following namespaces on Linux
  - **PID namespace** for process isolation
  - **NET namespace** for managing/separating network interfaces
  - **IPC namespace** for separating inter-process communication
  - **MNT namespace** for managing/separating filesystem mount points
  - **UTS namespace** for isolating kernel and version identifiers
    (mainly to set the hostname and domain name visible to the process)
  - **User ID (user) namespace** for privilege isolation

- user namespace must be enabled on purpose, it is not used by default
PID namespace

- allows to establish **separate process trees**
- the complete picture still **visible** from the **host** (outside the namespace)

```
root# docker run --rm -it debian/ps bash
root@3146c2faec9b:/# dash
# ps af
```

```
1029 ?  Ssl  7:48 /usr/bin/containerd
28834 ?  Sl  0:00 \_ containerd-shim -namespace moby ..........
28851 pts/0 Ss  0:00 \_ bash
28899 pts/0 S+  0:00 \_ dash
```

```
root@3146c2faec9b:/# ps af
```

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>STAT</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pts/0</td>
<td>Ss</td>
<td>0:00</td>
<td>bash</td>
</tr>
<tr>
<td>6</td>
<td>pts/0</td>
<td>S</td>
<td>0:00</td>
<td>dash</td>
</tr>
<tr>
<td>7</td>
<td>pts/0</td>
<td>R+</td>
<td>0:00</td>
<td>_ ps af</td>
</tr>
</tbody>
</table>
User ID (user) Namespace

- enables **different uid/gid** structures **visible** to the **kernel**

- **mapping** between uids in the namespace and “global” uids is **needed**

- by default, **root in the container is root in the host**

```
global (host) id’s
- 0
- 1
- ....
- 1000
- 1001
- ...
- 100000
- 100001

id’s in the namespace
- 0
- 1
```
Cgroups I.

- short for **control groups**
- they allow Docker Engine to **share available system resources**
- they implement **resource limiting** for different resources (CPU, disk I/O, etc.)
- they help to ensure that a single container **cannot** bring the system down
- cgroups are organized in a (tree) **hierarchy** for a given cgroup type
Cgroups II.

- a process (thread, task) **may be assigned** one cgroup
  - example of **memory control** (top level):
    - three children: web browsing (20 %), crypto mining (60 %), others (20 %)

- access via the /sys pseudo-filesystem is the simplest

  /sys/fs/cgroup/memory/ (top level)

  /sys/fs/cgroup/memory/web (specific cgroup)
Linux Kernel Capabilities

- capabilities turn the binary “root/non-root” dichotomy into a fine-grained access control system
- by default, Docker starts containers with a restricted set of capabilities
- Docker supports the addition and removal of capabilities
- additional capabilities extends the utility but has security implications, too
- a container started with --privileged flag obtains all capabilities
- running without --privileged doesn’t mean the container doesn’t have root privileges!
I am root. Or not?

- multiple levels of root privileges, from an unprivileged root user:
  - if user namespace is **enabled**, root inside a container has no root privileges outside in the host system
  - *by default*, root in a container has some privileges
    - but these are restricted by the **default set of capabilities**
  - we can **explicitly** add **extra capabilities** to our root in a container
  - with the **--privileged flag**, we have full root rights granted
root

root# docker run --rm -it debian/ip bash
root@b523a39fcc48:/# iptables -L -n
iptables: Permission denied (you must be root).
root@b523a39fcc48:/#

root

root# docker run --rm -it --cap-add=NET_ADMIN debian/ip bash
root@361c51aa11b0:/# iptables -L -n
Chain INPUT (policy ACCEPT)
target prot opt source destination

Chain FORWARD (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
root@361c51aa11b0:/#
Docker Daemon

- running containers (and applications) with Docker implies running the Docker daemon
- to control it, it requires root privileges, or docker group membership
- only trusted users should be allowed to control your Docker daemon
- it allows you to share a directory between the Docker host and a guest container
- e.g. we can start a container where the /host directory is the / directory on your host
Docker API

- An API for interacting with the Docker daemon
- By default, the Docker daemon listens for Docker API requests at a Unix domain socket created at `/var/run/docker.sock`
- With `-H` it is possible to make the Docker daemon listen on a specific IP and port
- You could set it to `0.0.0.0:2375` or a specific host IP to give access to everybody
- Docker API requests go, by default, to the Docker daemon of the host
Docker vs. chroot command

- a container **isn’t instantiated by the user** but the Docker daemon!
- anyone who’s allowed to communicate with the Docker daemon **can manage containers**
- that includes using any **configuration parameters**
- they can play with binding/mounting files/directories
- or decide which user id will be used in the container
  - including root (unlike eg. chroot)!
Escaping

● a **very general** term
● it does not necessarily mean **controlling the host system**
● **data access** (according to the C.I.A triad):
  ○ reading violating C.
  ○ modifying violating I.
● **executing** code **outside** the container (assigned cgroups and namely namespaces)
Escaping from/using Containers

- Methods:
  - Get access off the barriers (e.g. mounting filesystem while making a docker)
  - Inject a “hook” that is invoked by another party in the system
    - crontab rule, a kernel “notifier” running command on certain events
  - must run outside the container - APIs (e.g. inotify) won’t help
Examples of Docker-related incidents

- **unprotected access** to Docker daemon over the Internet
  - revealed by common Internet scans
  - instantiation of malicious containers used for dDoS activities

- **stolen credentials** providing access to the Docker daemon
  - used to deploy a container set up in a way allowing breaking the isolation
  - the attackers escaped to the host system
  - deployed crypto-mining software and misused the resources
Other kernel security features

- it is possible to **enhance Docker security** with systems like TOMOYO, AppArmor, SELinux, etc.
- you can also run the kernel with GRSEC and PAX
- all these extra security features require **extra effort**
- some of them are **only for containers** and not for the Docker daemon
- as of Docker 1.10 User Namespaces are **supported directly** by the Docker daemon
Docker Cheat Sheet I.

**start a new container from an image**
docker run IMAGE

**start a new container from an image with a command**
docker run IMAGE command

**start a new container in background**
docker run -d IMAGE

**start a new container and map a local directory into the container**
docker run -v HOSTDIR:TARGETDIR IMAGE
Docker Cheat Sheet II.

- *show a list of running containers*
  
docker ps

- *show a list of all containers*
  
docker ps -a

- *delete a container*
  
docker rm CONTAINER

- *start a shell inside a running container*
  
docker exec -it CONTAINER EXECUTABLE

- *stop a running container*
  
docker stop CONTAINER

- *start a stopped container*
  
docker start CONTAINER

- *download an image*
  
docker pull IMAGE
Practical Part
Cyber Range KYPO

- platform to organize and control cyber exercise, mostly CTF-like events
- set of services on the top of OpenStack cloud, providing separated sandboxes
  - machines are instantiated as VMs, connected using isolated network
- web portal mediating access to the environment and guiding participants through levels
  - description, tasks, hints
  - levels are linked using flags
- scoreboard and monitoring of progress for organizers
- platform is open-source, actively maintained by Masaryk University
  - https://kypo.muni.cz/
How To Get Started

● “book” your account at
  ○ [https://docs.google.com/spreadsheets/d/1gs2DPeYRO1gAdQS78D721GX5BAIrIG_WUKciKT1ua6Y/](https://docs.google.com/spreadsheets/d/1gs2DPeYRO1gAdQS78D721GX5BAIrIG_WUKciKT1ua6Y/)

● log in portal [https://isgc.crp.kypo.muni.cz](https://isgc.crp.kypo.muni.cz) using the booked credentials
  ○ you will start off the intro page
  ○ 16 “levels” in total (inc. intro etc.), each level contains
    ■ description
    ■ hints
    ■ specification of the flag
  ○ once you determine the flag, submit it to get to the next level

● interaction with VMs via either
  ○ embedded console (see the topology, click the “main” node (right mouse button) and open the console
  ○ directly using SSH (but ignore the “Get SSH Access“)
Train@main$

Task 0 & Task A

ssh -p XXX root@IP

Task B

Task C

SSH via Internet

OR

Browser console
Thank you for your attention.

Please be so kind and fill in our short questionnaire:

https://forms.gle/7kpR5gdE3L3bom8m6