Container Security:
What Could Possibly Go Wrong?

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 processes
- there are different containerization technologies available (Docker, Podman, Singularity, LXD, ...)
    - in this tutorial, we will focus mainly on Docker

Docker Terminology

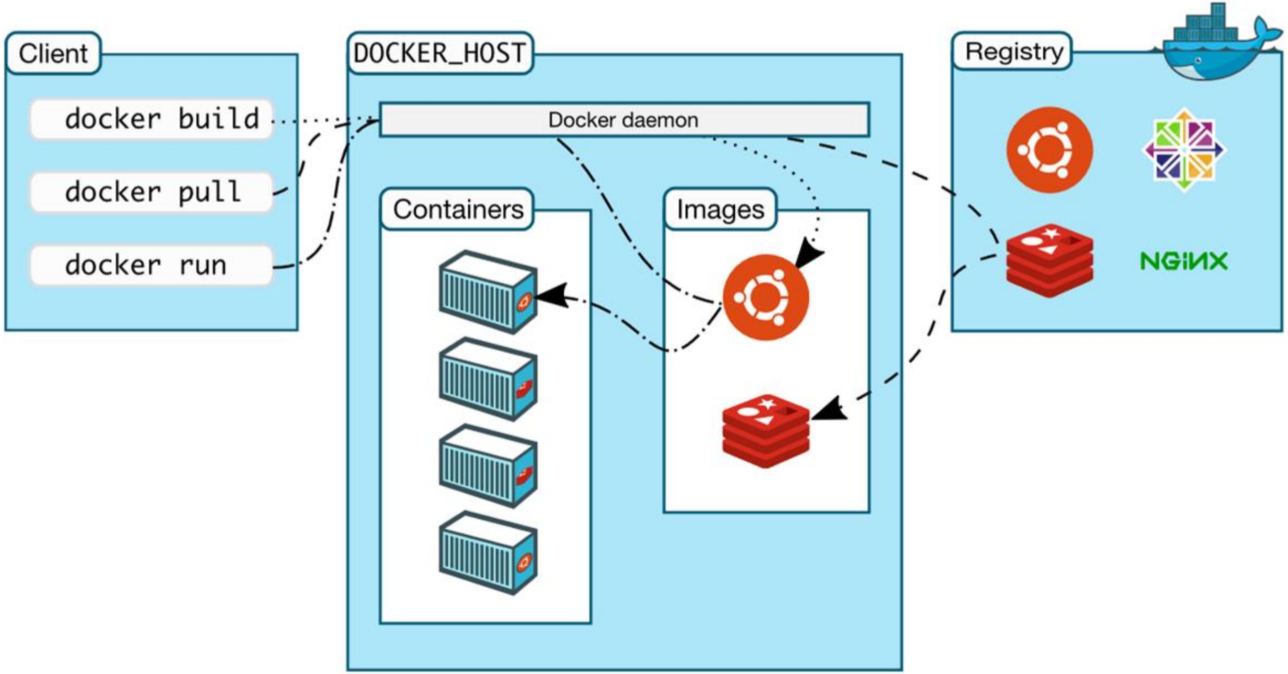
- **Docker container image** - a standalone package of files, which includes everything needed to run an application
  (code, runtime, system tools, system libraries and settings)

- an image is usually pulled from a **registry** to a host machine
  *(e.g. **DockerHub**)*

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

- a host machine runs the **container engine** (**Docker Daemon**) and manages individual containers

Docker Architecture



https://docs.docker.com/get-started/overview/

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/overlay2/51415bc9cd3ab2c47d218a897516ea2bf0545595fadf4a167ed5cfd3230a5f99/

- changes to the directory **are visible** from both sides (host and container)

- when the container is removed, any changes to its state **disappear** unless “committed” via **dockerd**

Starting 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**

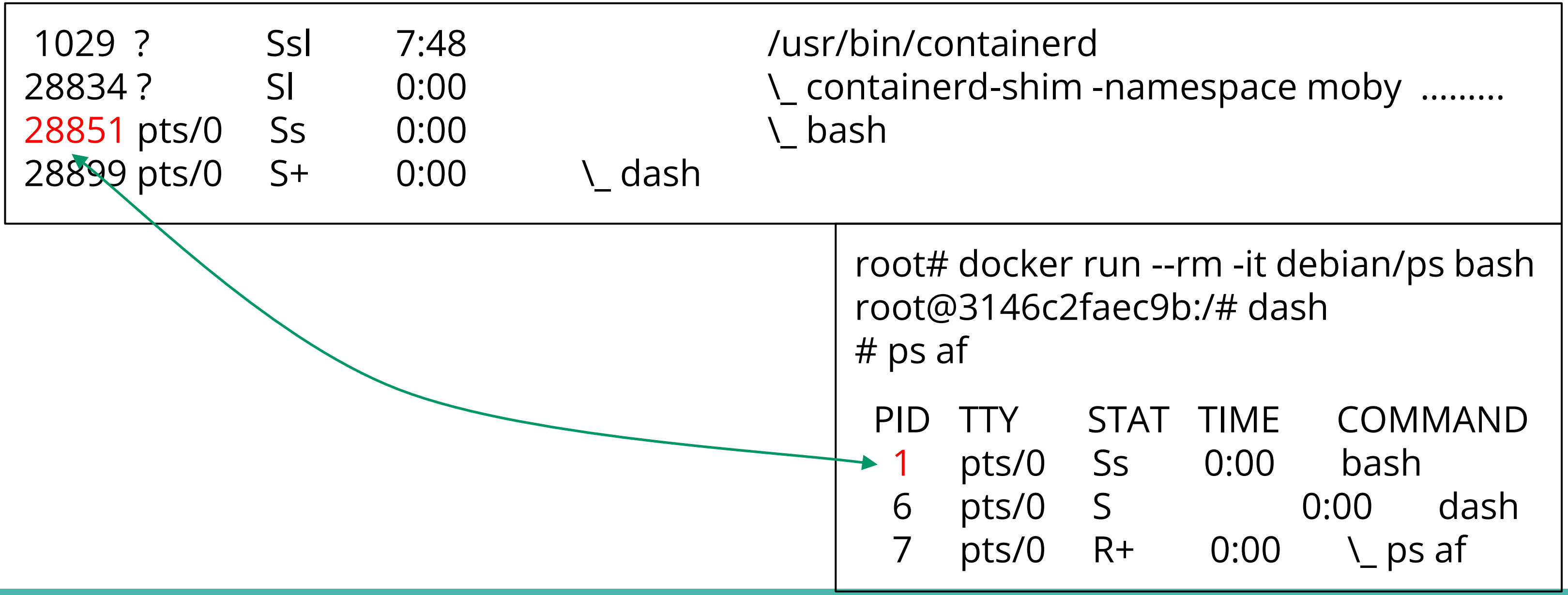
Namespaces

- 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 domainname 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)

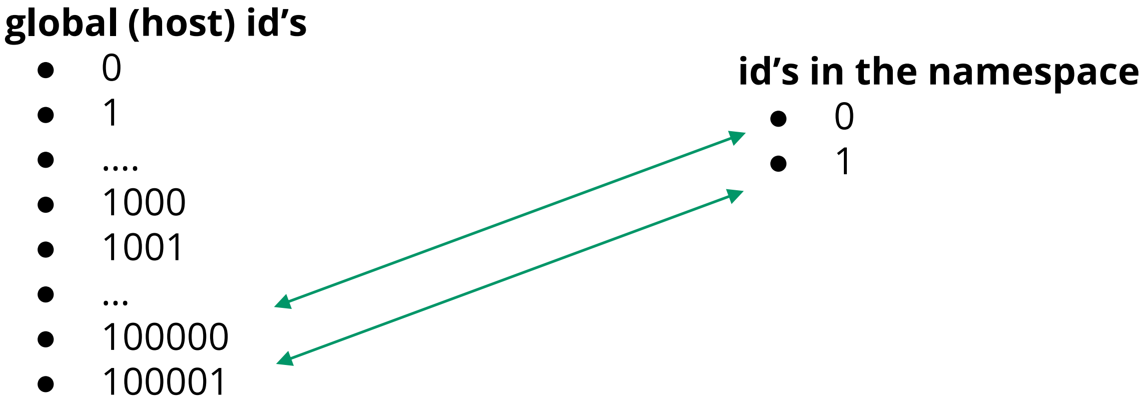


```
 1029  ?          Ssl      7:48                         /usr/bin/containerd
28834 ?           Sl       0:00                         \_ containerd-shim -namespace moby  .........
28851 pts/0       Ss       0:00                         \_ bash
28899 pts/0       S+       0:00              \_ dash
```

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

 PID   TTY      STAT   TIME      COMMAND
  1    pts/0    Ss      0:00      bash
  6    pts/0    S              0:00      dash
  7    pts/0    R+      0:00      \_ ps af
```

User ID (user) Namespace

- enables **different uid/gid** structures **visible** to the **kernel**
- **mapping** between uids in the namespace and “global” uids



global (host) id’s

- 0
- 1
- ….
- 1000
- 1001
- …
- 100000
- 100001

id’s in the namespace

- 0
- 1

- by default, user namespace is not enabled by Docker, i.e. **root in the container is root in the host** !

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
  - access via the /sys pseudo-filesystem is the simplest (/sys/fs/cgroup)
- Setting up a cgroup

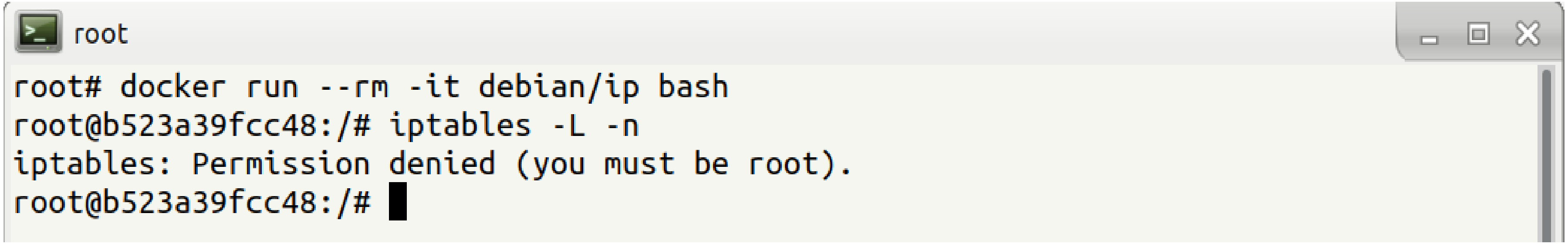
```
# create a specific cgroup:

mkdir /sys/fs/cgroup/memory/web

# manipulate with the cgroup parameters using file in /sys/fs/cgroup/memory/web

# enter the new cgroup with the current shell  to apply to limit:

echo $$ > /sys/fs/cgroup/memory/cgroup.procs
```

Linux Kernel Capabilities

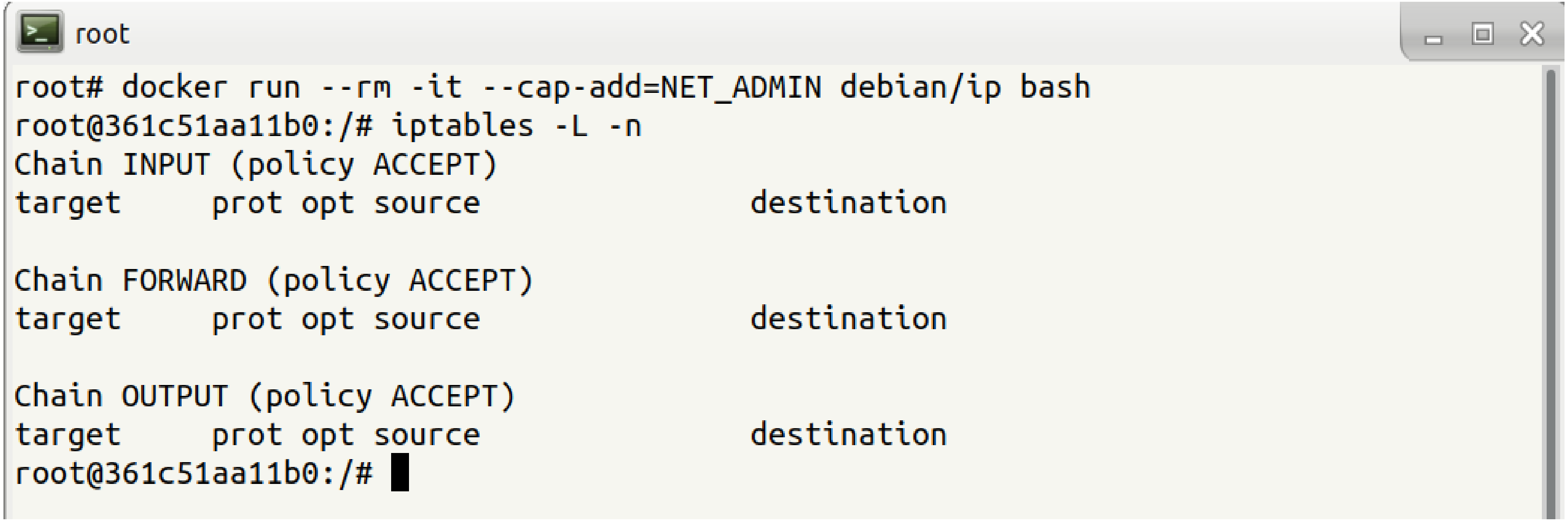
- capabilities turn the binary “root/non-root” dichotomy into a **fine-grained access control system**
- by default, Docker starts containers with **a restricted set of capabilities**
- Docker supports the **addition** and **removal** of capabilities
- additional capabilities extends the utility but 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 the root. Or not?

- multiple levels of the root privileges, from an unprivileged root user:
  - if user namespace is **enabled**, the root inside a container has no root privileges outside in the host system
  - **by default**, the 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# docker run --rm -it debian/ip bash
root@b523a39fcc48:/# iptables -L -n
iptables: Permission denied (you must be root).
root@b523a39fcc48:/# 
```



```
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 the **root privileges**, or a **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 a 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**

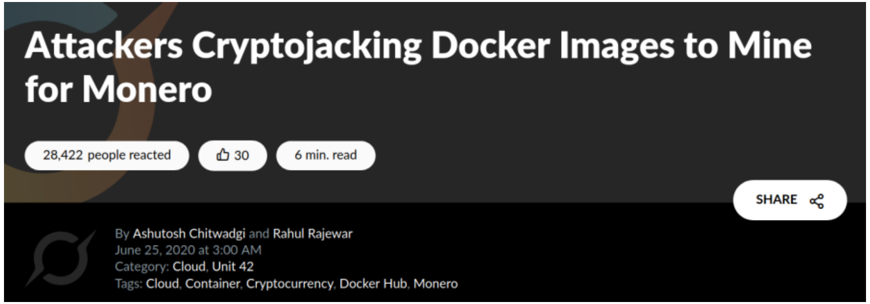
Docker vs. chroot command

- a container **isn’t instantiated by the user** but the Docker Daemon!
- anyone who is 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) !

Container Security

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



Attackers Cryptojacking Docker Images to Mine for Monero

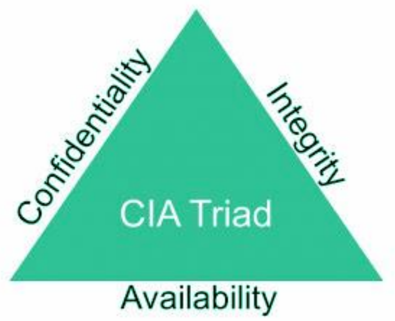
By Ashutosh Chitwadgi and Rahul Rajewar
June 25, 2020 at 3:00 AM
Category: Cloud, Unit 42
Tags: Cloud, Container, Cryptocurrency, Docker Hub, Monero

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

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

- 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
  - an 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

Cheat Sheets

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 *stop a running container*

docker ps

docker stop CONTAINER

show a list of all containers

start a stopped container

docker ps -a

docker start CONTAINER

delete a container

download an image

docker rm CONTAINER

docker pull IMAGE

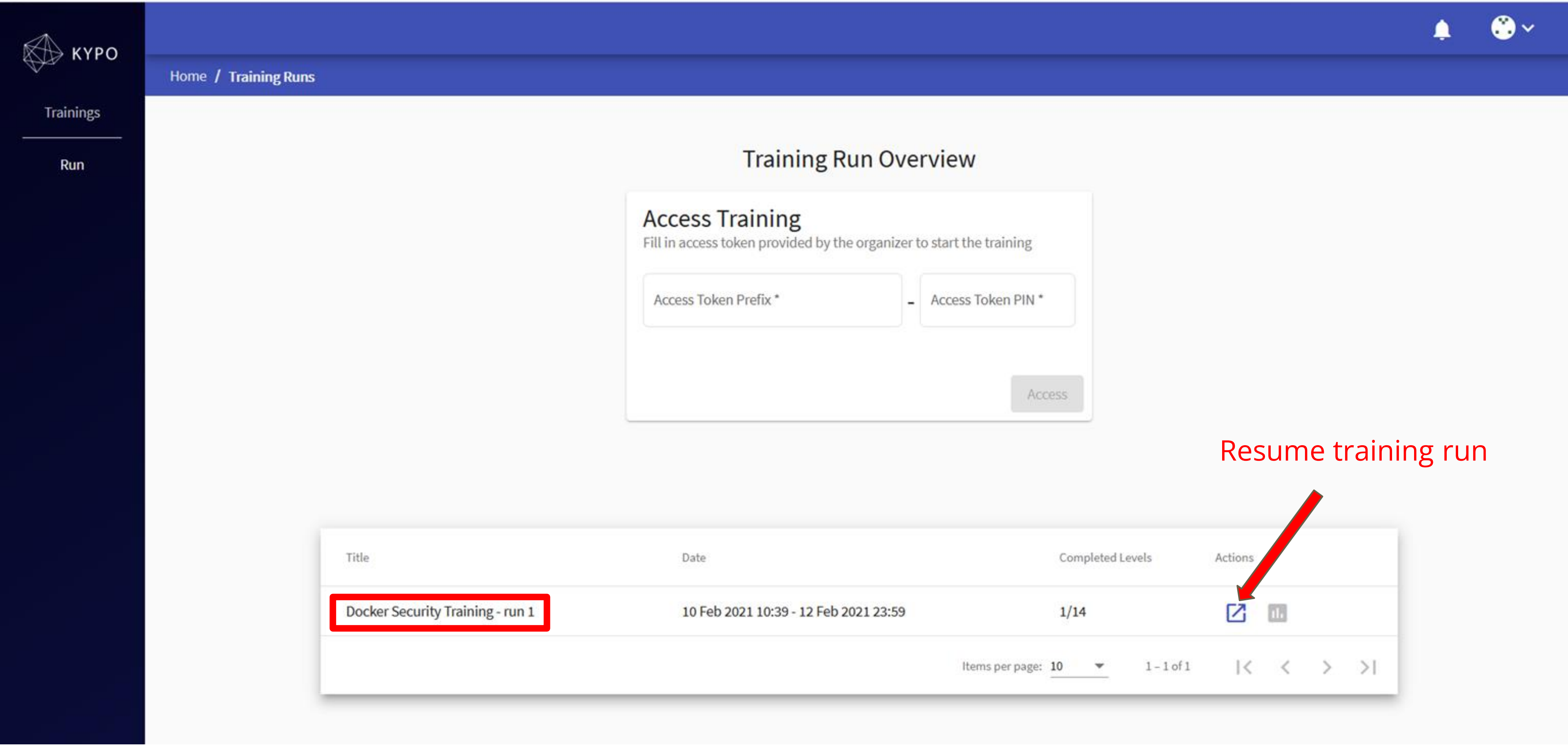
start a shell inside a running container

docker exec -it CONTAINER EXECUTABLE

Practical Part

Accessing the environment

- Two things:
    - access to the web portal
    - access to the virtual machines
- Portal access
    - Book an account at
        - https://docs.google.com/spreadsheets/d/1xo2TBjov7YU_HaHrLjW4b4ZQI0Bg5cv47owYI1GldP8
        - Just pick up a free line and put your name there (or other identifier of yours).
- Visit and login with the booked credentials.
    - Use “**Login with local issuer**”



Training Run Overview

Access Training

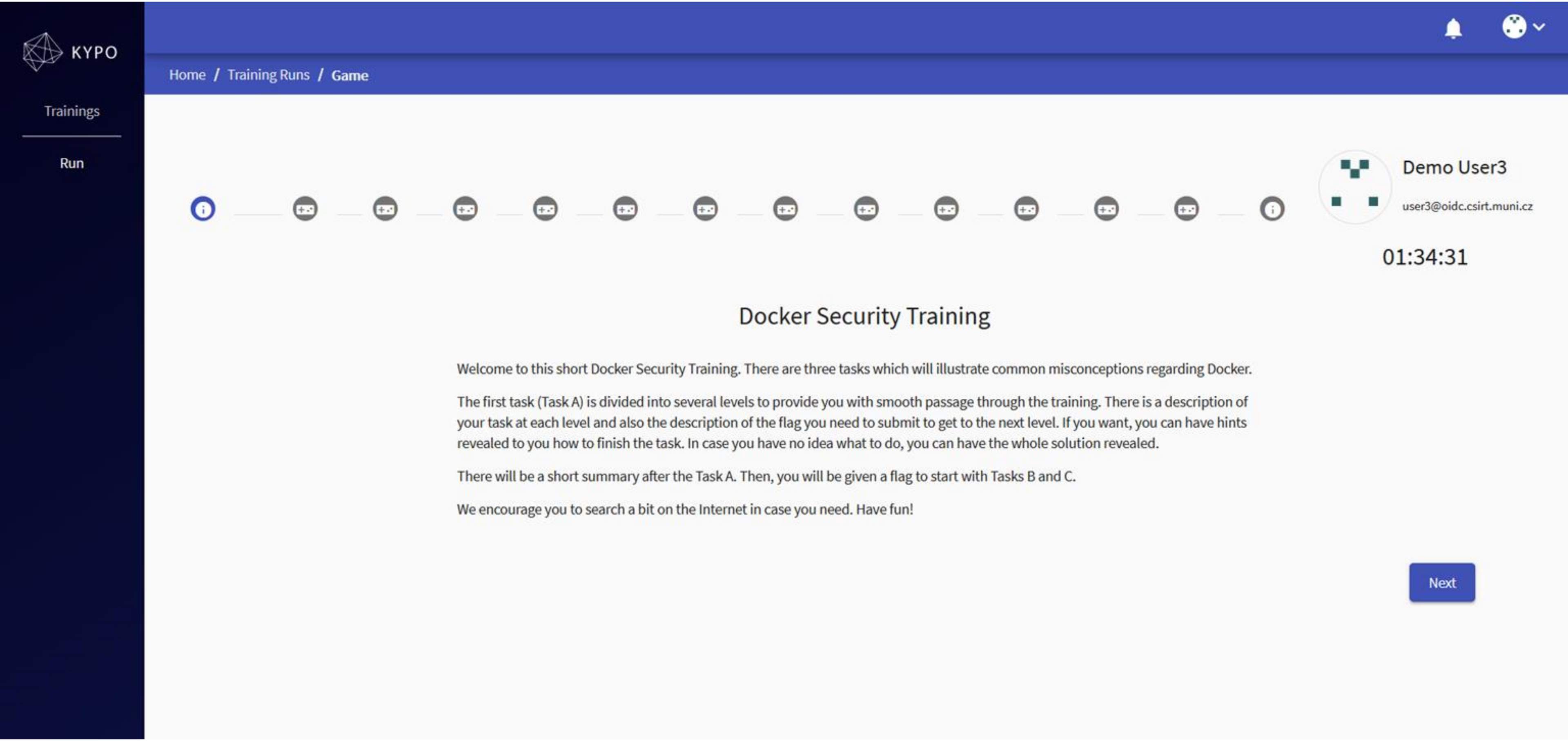
Fill in access token provided by the organizer to start the training

Access Token Prefix * - Access Token PIN *

Title	Date	Completed Levels	Actions
Docker Security Training - run 1	10 Feb 2021 10:39 - 12 Feb 2021 23:59	1/14	

Items per page: 10 1 – 1 of 1

Resume training run



Docker Security Training

Welcome to this short Docker Security Training. There are three tasks which will illustrate common misconceptions regarding Docker.

The first task (Task A) is divided into several levels to provide you with smooth passage through the training. There is a description of your task at each level and also the description of the flag you need to submit to get to the next level. If you want, you can have hints revealed to you how to finish the task. In case you have no idea what to do, you can have the whole solution revealed.

There will be a short summary after the Task A. Then, you will be given a flag to start with Tasks B and C.

We encourage you to search a bit on the Internet in case you need. Have fun!

Access to machines

- SSH access
  - ssh -p **PORT** training@demo.crp.kypo.muni.cz
    - the **PORT** and the password is available from the sheet with booked credentials
- Console access
  - From the portal you can use the embedded console
  - See the visualised topology, found the "main" machine (10.20.30.100) and select "Open console".