# Linux Advanced Writeup

## 0. Foreword

In order to pass my exams I kinda have to study but its just something I just don’t do.. I’m the type of person who doesn’t care about studying and just likes to do put everything into practical use and that’s what I’ll do for this writeup of Linux Advanced.

In here I’ll write down all commands with screenshots, code and explanations in order for other people and myself to learn from it.

## 1. Introduction

In this course we will go over:

* Docker
* Inodes and linking
* Network management
* Package management
* Memory management
* Disk management
* Scheduling
* Logging
* SSH, scp, VNC

But mostly Docker

## 2. Docker

### 2.1 Docker: What is docker and installation

#### 2.1.1 What is docker?

Docker is an open source framework which makes it possible for an application to be placed in a lightweight moveable container.   
You can easily more this container between platforms as long as the Docker engine is being run on it!

Docker prevents the overheat that results in using multiple virtual machines that need their own OS, which not only spares resources bit applications work much faster due to not requiring a hypervisor!

Docker engine is a tool that works with 3 technologies:

* Namespaces
* Cgroups
* Capabilities

We will see more about this later

A docker container makes use of a kernel so Linux containers can’t work in a docker environment from Windows.

Before 1 physical machine was used for each application:

* This was an Apache server + Nginx server
* This gave problems with libraries and dll-files
* Much of the server resources was never used
* Many apps in the same OS don’t work well
* You couldn’t run Linux and Windows on the same server

After that we used Hypervisor-virtualization, multiple virtual server per physical machine:

* This made better use of the servers resources
* A lot of overheat due to having to run a separate OS for each application
* You could now run multiple Windows and Linux distributions

Now we have Container-Virtualization, an application in a runtime environment:

* Its lightweight and it uses less resources than a VM
* It uses one physical machine with only 1 OS that can run tens or more containers with their own application.

When we used virtual machines we had an infrastructure which is the server and its hardware, on top of that we had our Hypervisor, mostly Windows and Linux and inside of our hypervisor we where then running multiple virtual machines each running their own Operating system with the application on top.

Graphical user interface

Description automatically generated

With docker we have our infrastructure and on top of that our operating system, on our operating system we have docker and docker will hold all the containers with the applications inside.

Chart

Description automatically generated with medium confidence

There are different docker components to take note of:

* Docker Engine
  + Also named: Docker Daemon and Docker Runtime
  + Responsible for working with root-filesystem
* Docker Image
  + Defines OS of the container
  + run a docker image with   
    | docker run -it ubuntu /bin/bash
  + See all your images with  
    | docker images
  + Download an image  
    | docker pull <ubuntu>
* Containers
  + Gets build with downloaded images
  + Running copy of images
  + start with  
    | docker run <hello world>
* Repository
  + Images are found and found from here
  + there are standard repo’s for Ubuntu, ngrix, …
  + Different repo versions available
* Registry
  + Find docker registry at hub.docker.com
  + Repo’s are held in the registry
  + In the Ubuntu repo there are different Ubuntu versions

#### 2.1.2 Docker installation

For commands we will go through a little faster because a lot has been touched on in my Linux Essentials writeup

Go in as root  
| sudo su

Download docker (If you get a notification that hashes mismatch you will just have to try again)  
| wget -qO- https://get.docker.com | sh

Give normal users permission to use docker so you don’t have to give root access with  
| usermod -aG docker daan

when we did this we can leave root with   
| exit  
And see if docker has installed correctly with   
| docker version

We can see the rights of the user at docker.sock  
| ls -l /var/run/docker.sock

We can already download an Ubuntu dock upfront so we can use it later  
| docker pull ubuntu

But we can also run docks before we pulled them, docker will then automatically pull and run the container  
| docker run hello-world

If we now want to see all active containers we can use  
| docker ps

If we want to see all containers currently on the system we can use  
| docker ps -a

We can see all of our downloaded images with docker images  
| docker images

if we put --help behind one of these commands we can get more info  
| docker images help

### 2.2 Docker Hub & Images

#### 2.2.1 Docker Hub

Docker hub is ofcorse where the images are being downloaded.  
We can check out https://hub.docker.com/ and check out Explore

Graphical user interface, text, application, email

Description automatically generated

We can also search explore through the terminal, for example we are looking for an Ubuntu image  
| docker search ubuntu

Graphical user interface, text, application, email

Description automatically generated

We can then search for ubuntu and choose the rolling version  
In this example we will go for Ubuntu 16.04

If we now want this Ubuntu version we can go in Linux and do  
| docker pull ubuntu:16.04

#### 2.2.2 Docker images

If we done that we can see our new downloaded ubuntu image with  
| docker images

When we don’t add a version behind it will pull the latest version  
| docker pull ubuntu

We can also remove our images with  
| docker image rm ubuntu:16.04

If we want to remove all images that aren’t currently in use we use  
| docker image prune -a

We can also save an image in a tar file  
| docker save -o ubuntu.tar ubuntu

If we then remove our ubuntu image  
| image rm ubuntu  
  
We can easily load our ubuntu image again with  
| docker load -i ubuntu.tar

### 2.3 Working with containers

We can simply run ubuntu with   
| docker run ubuntu

But lets say we want to send a command directly after run  
| docker run ubuntu /bin/echo “Hello World”  
| docker run ubuntu:14.04 cat /etc/os-release  
| docker run ubuntu /bin/sleep 20

We can also detach the console from the container after running a command so the console doesn’t have to wait  
| docker run -d ubuntu /bin/sleep 20

we can also enter the bash itself with  
| docker run -it ubuntu /bin/bash  
You can use exit or CTRL + P + Q  
| exit

We can get access again to the shell with docker attach <CONTAINER ID>  
| docker attach 16e1985d16c7

we can find all active containers with  
| docker ps  
| docker container ls

Or we can ofcorse check for all containers with  
| docker ps -a  
| docker container ls -a

We can also remove a container with rm   
| docker container rm 16e1985d16c7

We can give another name so we don’t have to use the id  
| docker run --name 16e1985d16c7 PostgreSQLContainer

And ofcorse rename the container  
| docker container rm PostgreSQLContainer PostgreSQLC

We can pause a container with  
| docker container pause PostgreSQLC

And resume the container with  
| docker container unpause PostgreSQLC

Stop a container with  
| docker container stop PostgreSQLC

Start a container with  
| docker container start PostgreSQLC

Get the stats with  
| docker container stats PostgreSQLC

And logs with  
| docker container logs PostgreSQLC

We can throw al containers that aren’t running away with  
| docker container prune

To remove a specific container we can do   
| docker rm PostgreSQLC

If the container is still running you can’t instantly throw it away we can force it with  
| docker rm -f PostgreSQLC

### 2.4 Containerdata, Layers & OverlayFS

#### 2.4.1 Data stays in your container

Ofcorse when we add data to a container the data stays when we exit the console.

We can try this with he following commands  
| docker run -it ubuntu /bin/bash  
| echo Hello > /tmp/testfile  
| exit

We have now created a file inside of our Ubuntu   
If we now use   
|docker diff ubuntuContainer  
We can see the changes done to the filesystem  
Now we can start the container again with the prompt  
| docker start -i UbuntuContainer  
We can now still see that the file still exists after we restarted it  
| ls /tmp

#### 2.4.2 Images layers

A docker image is build up on multiple image layers via the  
Overlay2-filesystem, this works with union layers

Image layers are build up out of:

* Bottom layer: The rootfs from the OS
* Middle layer(s): the application
* top layer(s): updates and bugfixes

These layers are stacked on top of each other aka. Union mounts, they are transparent for the end user

During runtime when a container starts on top of the layers one more gets placed that monitors all additions and changes to the filesystem (This is the only accessible layer)

Graphical user interface

Description automatically generated

#### 2.4.3 Image and container data

Anywhere where we use the container id we can use the shortid like UbuntuContainer and the other way around! We can also choose between the image name and image shortid

If we want to inspect our container we can use  
| docker inspect UbuntuContainer

If we want to inspect an image we can use  
| docker inspect ubuntu

We can also get data from the docker on your machine with  
| docker info  
With this we can see what containers we have are active, paused and stopped and images currently stored

### 2.5 Processes, Logs, Cgroups & Namespaces

#### 2.5.1 Introduction

We can also monitor ongoing processes in a running container we can test this by starting a process in a container  
| docker run -d ubuntu bash -c “ping -c 50 8.8.8.8”

As we are pinging 50 times to google we can monitor it through  
| docker ps  
We can also monitor one specific container through the shortid or name  
| docker top UbuntuContainer  
We can also see the console logs from a container with  
| docker logs UbuntuContainer

Each container can have its own root file system, profess-tree, etc! All while the container is completely isolated, this is possible thanks to the containers having each their own namespace!

Containers make use of Cgroups in which they can call for resources. with this limits can be placed and each container has its own cgroup. Cgroups are really flexible and prevent other containers to crash when one container is hit with a DDOS.

Containers also make use of capabilities. This makes it possible to give containers their own rights.

#### 2.5.2 Limits

We can also limit the access to resources for containers!

We can for example say that the alpine image we run can only take 100Mb memory, we also name it mb100 for easier reference  
| docker run -d --name mb100 --memory 100m alpine top

we can then check the stats for the memory  
| docker stats --no-stream

We can also assign CPUs to a docker container  
| docker run -d --name 1cpu --cpuset-cpus 0 benhall/stress

Or assign multiple CPUs at once  
| docker run -d --name 1cpu --cpuset-cpus 0,3 benhall/stress

There are 3 ways to define CPUs:

* --cpu-shares (of -c) : the number opposite to the standard value of 1024
* ---cpus=0.000 : number of CPUs. 0.000 means no limit
* --cpuset-cpus=”” : CPUs in which to allow execution

#### 2.5.3 Namespaces

When starting a container a network interface gets automatically created or defined. This makes it so a container gets an unique   
IP-address.

We can check this with  
| docker run -it alpine addr show

When we change the network-namespace to “host” the containers network won’t be isolated to its own interface and the   
container-process gets access to the host machine network

We can do this via  
| docker run --net=host alpine ip a

A container can only see the processes that are inside the same Pid namespace

Each container will by default run its own Pid namespace, so only the processes in its own container will be seen.

When we change the namespace to host the container will see all processes that are running on the host  
| docker run --pid=host alpine ps aux

Sometimes giving other containers access to another is sometimes needed for debugging tools but will be seen as bad practice security wise!

We can start with an nginx container and add a new network and process namespace   
| docker run -d --name http nginx:alpine

Other containers can share these names by accessing them in order to do this we can do   
| docker run --net=container:http centos curl -s localhost

We can share the Pid-Namespace in the same way  
| docker run -d --name webserver nginx:alpine

We can also start a new container in the same Pid-namespace so this container can see the other container  
| docker run --pid=container:webserver alpine ps aux

But watch out! Containers might have to much permissions inside of each other for example like this  
| docker run –pid= container:http-server alpine killall nginx

### 2.6 Shell access in a container

We will now go over getting shell access in a docker application

We start with pinging 300 times to Google’s DNS  
| docker run -d ubuntu “ping -c 300 8.8.8.8”

We can then check for the pid of our container for this we first have to go to  
| docker ps  
And after that we can use that name to find the pid of the container  
| docker inspect tender\_shamir | grep -i pid

To then gain access we use nsenter with the namespaces (-m -u -n -p -i ) and -t for target  
| sudo nsenter -m -u -n -p -i -t 26528

Most container won’t run a bash or ssh like the one we just used but it does have an app for which the container is started so we can simply use  
| docker-enter 16e1985d16c7

If we then want to exit again we need to use  
| docker exec -it 16e1985d16c7 /bin/bash  
And if docker exec doesn’t exists  
| docker run --rm “/usr/local/bin:/target jpetazzo/nsenter”

### 2.7 Dockerfile and pushing to Docker Hub

#### 2.7.1 Introduction

We always build a docker image via a Dockerfile

Docker-file:

* Always named Dockerfile
* plaintext file
* Simple layout
* holds image build instructions
* to build the image use  
  |docker build

#### 2.7.2 Creating a Dockerfile

We are going to make an ubuntu docker image that has access to the ping command

First we make and enter a directory named   
| mkdir Dockerbuild && cd Dockerbuild

Then we create a file with vim and edit it   
| vim Dockerfile

Inside this file we add the following  
| # ubuntu based Dockerfile Image  
| FROM ubuntu:18.04  
| LABEL maintainer=”mail.mail@hereyourmail.mail”  
| RUN apt update && apt -y upgrade && apt -y install iputils-ping  
| CMD echo “BITCONNEEEEEEEEEEECT”

We can then build the image  
| docker build -t testimage:0.1

Now we can test it by using  
| docker run testimage:0.1

And now if we enter bash we can make use of the ping command  
| docker run -it testimage:0.1 bash

If we want to see what actually happened in the image right now we can check it with  
| docker history testimage:0.1

An entrypoint can’t be overridden during runtime except if   
--entrypoint is added to the command.  
The contents of the CMD command (if there) will be added as parameter to the entrypoint command.  
The command behind the run command will also be given as a parameter.

For this we use a slightly different Dockerfile  
| FROM ubuntu:18.04  
| LABEL maintainer=[mail.mail@hereyourmail.mail](mailto:mail.mail@hereyourmail.mail)  
| CMD [“Hello World”]  
| ENTRYPOINT [“echo”]  
  
Then we build again  
| docker build -t test .

We can then try to run it  
| docker run test  
| docker run test hellow

When we now do   
| docker run -it test /bin/bash  
/bin/bash will be just echoed   
If we now want to enter the command prompt we can use   
| docker run -it --entrypoint=”/bin/bash” test

ENV can also be used to set environment variables in a container

For this we once again use a slightly different Dockerfile this time we define some variables  
| FROM ubuntu:18.04  
| LABEL maintainer=”mail.mail@hereyourmail.mail”  
| ENV var1=ping var2=”-c 5” var3=8.8.8.8  
| CMD $var1 $var2 $var3

We build it  
| docker build -t pinger .

and run  
| docker run pinger

We can also add files to an image with copy  
For this we can make a small .c script named helloworld.c  
| #include <stdio.h>  
| void main(){  
| printf(“Hello world!\n”);  
| }

Then we create another Dockerfile  
| FROM frolvlad/alpine-gcc  
| LABEL maintainer = ”mail.mail@hereyourmail.mail”  
| COPY helloworld.c /root  
| RUN gcc -o helloworld helloworld.c  
| CMD [“./helloworld”]

And then we can test it  
| docker run helloworld-app

We can use USER to start a container with an user, this user has less permissions than the default (root)

For this once again a new Dockerfile  
| FROM fedora  
| # USER root  
| RUN groupadd -r noprivileges && useradd -r -m -g noprivileges -s /sbin/nologin -c “Demo user” normaluser && echo normaluser:hi | chpasswd && usermod -aG wheel normaluser  
| RUN dnf install -y tree iputils  
| # chmod on ping to allow to run for unprivileged users  
| RUN chmod 4755 /bin/ping  
| USER normaluser  
| WORKDIR /home/normaluser  
| RUN touch Welcome.txt  
| CMD ping -c10 8.8.8.8

we can now build this image  
| docker build -t userdemo .

There is a lot we can try so I’ll sum it up  
| docker run -it userdemo bash  
| ls  
| ls /root  
| sudo ls /root  
| pwd  
| tail -1 /etc/passwd  
| tail -1 /etc/group  
| whoami  
| su - normaluser  
| exit  
| docker run userdemo

#### 2.7.3 pushing an imager to docker hub

ofcorse we can also add our own Images to docker hub, this goes as follows.   
  
First you need to make an account on docker, if you got this far I assume I don’t need to hold your hand for this.

Then we will go to create Repository, the namespace is already filled in with our username, we name our repo testimage and description “my first repo” visibility is public cause you have a limited amount of private storage and then we press create.  
  
We can first look for a list of our images  
| docker images  
  
We can then give it a tag using our username  
| docker tag testimage:0.1 <username>/testimage:1.0

Now we can see our image with its new tag  
| docker images  
  
Ofcorse in order to push to our account in docker hub   
| docker login  
  
Then we will push it to the hub  
| docker push <username>/testimage:1.0  
  
Then we log out again and remove our local image  
| docker logout && docker rmi <username>/testimage:1.0  
  
Now we will run our image again so we can test if it works, this will be pulled from docker hub again cause we just removed it  
| docker run <username>/testimage:1.0

### 2.8 Container volumes

There are different types of volumes:

* Anonymous volumes: A volume assigned to the directory by docker itself  
  | docker run -v /path/in/container  
  We can delete these together with the container  
  | docker rm -v container
* Named volumes: docker also assigns these but in this case you give the volume its name  
  | docker volume create somevolumename  
  | docker run -v somevolumename:/path/in/container
* Host volumes: This is a self chosen directory on the host that’s mapped with a directory in the container  
  | docker run -v /path/on/host:/path/in/container  
  | docker run -v /path/to/file/on/host:/path/in/container

Some examples:

* Anonymous volumes  
  | docker run -it -v /containervol ubuntu /bin/bash  
  Text

  Description automatically generated
* Named volumes  
  | docker volume create myvolume  
  Graphical user interface, text

  Description automatically generated
* Host volumes  
  | mkdir hostvol  
  | echo “some text in a file” > hostvol/myfile  
  | docker run -it -v /home/student/hostvol/:/containervol ubuntu /bin/bash  
  Text

  Description automatically generated

We can have multiple containers use the same volume

In order to do this we can do  
| docker run -it -v /tmp/vancontainer:/test-volume   
--name=voltainer1 ubuntu /bin/bash  
| echo “hello world” > /test-volume:myfile  
| CTRL-P-Q  
  
| docker run -it -v /tmp/vancontainer:/test-volume   
--name=voltainer2 ubuntu /bin/bash  
| cat /test-volume/myfile

There is also another way to share volumes  
| docker run -it -volumes-from=voltainer2  
--name=voltainer3 ubuntu /bin/bash  
| cat /test-volume/myfile

If we want to create a docker volume we can simply do  
| docker volume create myvol

To use a volume we use -v or --volume   
| docker run -it -v myvol:/app --name=volumetest ubuntu /bin/bash

The same goes with --mount   
| docker run -it --mount source=myvol,target=/app   
--name=mounttest ubuntu /bin/bash

To delete a container we have different commands

Deleting a container without volumes  
| docker rm containerName

Deleting a container with an anonymous volume  
| docker rm -v containerName

Deleting a named volume  
| docker rm containerName  
| docker volume rm volumeName

Deleting a host volume  
| docker rm containerName  
| rm -rf hostDirectory

### 2.9 Container networking

#### 2.9.1 Docker0-bridge

Networking is being maintained by the Docker0-bridge

When we do  
| ip a  
We can see our docker0 interface

If we want to work with this we will first have to install bridge-utils  
| sudo apt install bridge-utils

Now we can already check for interfaces, we might not see them yet  
| brctl show docker0

Just like before we can use a Dockerfile to test docker networking  
| mkdir networking  
| cd networking  
| vim Dockerfile

| # Image for network testing  
| FROM ubuntu:18.04  
| LABEL maintainer=”mail.mail@hereyourmail.mail”  
| RUN apt update && apt -y install iproute2 iputils-ping dnsutils traceroute

| docker build -t networking .  
| docker images

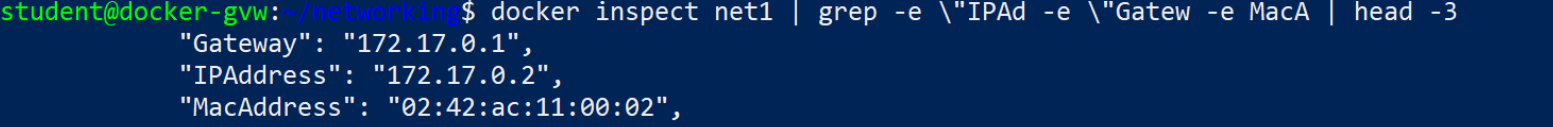
We can now test our networking  
| docker run -it --name net1 networking  
| ip a  
| ip route  
| CTRL P Q  
| ip a s docker0  
| brctl show docker0  
| exit  
We will only see one interface

| docker run -it --name net2 networking  
| ip a  
| ip route  
| CTRL P Q   
| brctl show docker0  
Now we will see 2 interfaces

We can now do   
| docker attach net1  
| ip a  
| ping -c 8.8.8.8  
| ip route  
| CRTL P Q  
To check how our network is functioning right now

“

As w are out of the container now we can do   
| docker inspect net1  
So we can see some more data about our network



#### 2.9.2 Config files

There are also config files we can look at  
| docker inspect net1 | grep resolv.conf

Or we can also do  
| sudo cat /var/lib/docker/containers/<containerid>/resolv.conf

This contains multiple things:

* Nameserver information
* This is a copy of the docker host en is created when starting the container
  + copies the file that’s inside the image
  + Can be changed during runtime but will be reverted on restart
* Can be changed by adding arguments in the run command
  + | docker run -it -dns 8.8.4.4 --name net3 networking
  + CTRL P Q
    - |docker inspect net 3 | grep -wA2 Dns

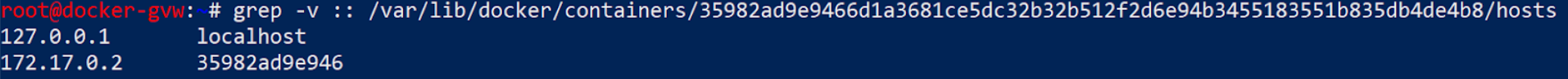
Graphical user interface, text

Description automatically generated

We can also check for the host   
| docker inspect net1 | grep host

There is also  
| sudo cat /var/lib/docker/containers/<containerid>/hosts

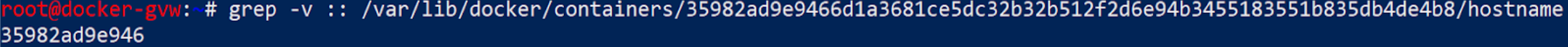
* contains the ipv4 and ipv6 dns entries
* overwrites the file that’s inside the image
  + this can also be changed when the container is running
  + Arguments can be added to the run -commando  
    | docker run -it –add=<name:op> --add-host=<name:ip>   
    --name=net4 networking



We can repeat this for the hostname  
| docker inspect net1 | grep hostname

the same can be done with cat  
| sudo cat /var/lib/docker/containers/<containerid>/hostname

* Contains the name of the container
* We can also override this in the run-commando
  + docker run -it --hostname=<hostname> --name=net5 networking



#### 2.9.3 Port forwarding

We will now make an image to test port forwarding with   
| mkdir myapache  
| cd myapache  
| vim Dockerfile  
 | # Image for port forwarding  
 | From networking  
 | LABEL maintainer=”mail.mail@hereyourmail.mail”  
 | RUN apt update && apt -y install apache2  
 | EXPOSE 80  
 | ENTRYPOINT [“apache2ctl”]  
 | CMD [“-D”, “FOREGROUND”]

| docker build -t myapache .  
| docker images

then we can run again  
| docker run -d -p 5001:80 --name web1 myapache   
| docker ps   
Now we can see the ports that port 5001 on the docker host has poft forwarded to the 80 container

We can also navigate through a web browser to a containers website  
through docker host we can do   
| lynx localhost:5001  
If you are on a vm you can also access it outside your vm:

* VMware : https://<dockerhost-ip>:5001

| docker inspect web1 | grep -C3 80/tcp  
| docker port web1

Port forwarding can be done through different methods  
  
| docker run -d -p <host-id>:5002:80 --name=web2 myapache

* You can redirect a certain ip address to a port within the container
* you send to port 5002 to the specified <dockerhost-ip> and then you’ll reach the container on port 80

| docker run -d -P --name=web3 myapache

* You can forward all exposed ports via the option -P
  + These ports then become variable ports
* docker port web3 shows which ports are forwarded on the docker host
  + 80/tcp -> 0.0.0.0:32768

| docker run -d -p 80 –name=web4 myapache

* You can also forward one exposed port in the Dockerfile with the option -p <port>
* to show which port his being forwarded
  + | docker port web4

#### 2.9.4 Communication between containers

We can add an extra network to certain containers to communicate which each other  
| docker network create --subnet 172.25.0.0/24 isolated\_nw

Each container gets by default an ip in the docker0-bridge  
| docker run -itd --name contain 1 networking

The container gets an extra ip in the isolated\_nw network  
| docker network connect isolated\_nw contain1

the container gets only an ip in the isolated\_nw network  
| docker run -itd --network isolated\_nw --name contain2 networking

remove a container from a network  
| docker network disconnect isolated\_nw contain2

to remove the network  
| docker network rm isolated\_nw

## 3. Ubuntu