LABS Unidade 2

Utilize o Docker Cheat-sheet para ajudar com os comandos:

Você vai precisar de uma máquina Linux para esses LABS.

LAB 1

Objetivo: Entender o uso do chroot e pivot_root

1. Inicialize um diretório com o conteúdo de um sistema base linux
    1. Se você estiver em uma distribuição baseada em debian, como o ubuntu, siga esses passos:

    ```
    sudo apt install debootstrap
    sudo debootstrap --variant=buildd --include=iputils-ping,python3,procps,net-tool,iproute2 --arch=i386 stable ./rootfs
    ```

    2. Se estiver em uma distribuição baseada em rpm, como o fedora, siga os seguintes passos:

    ```
    sudo dnf -y \
    --installroot=$PWD/rootfs \
    --releasever=24 install \
    @development-tools \
    procps-ng \
    python3 \
    which \
    iproute \
    net-tools
    ```

2. Teste o funcionamento do chroot com o novo diretório

```
touch /tmp/host.txt
touch rootfs/tmp/rootfs.txt
sudo chroot rootfs

hash -r
ls /tmp/host.txt
ls /tmp/rootfs.txt
exit
ls /tmp/host.txt
```

3. Executando o pivot_root

```
# Crie um ponto de montagem para o container
mkdir new_root
# Crie uma namespace de mountpoints para ser usada daqui em diante
sudo unshare -m
# monte o rootfs para o novo mountpoint new_root
mount --bind ./rootfs/ new_root
# Verifique que o rootfs agora está montado no new_root
ls new_root/tmp/rootfs.txt
# crie um ponto de montagem para abrigar o atual root da máquina no novo namespace de montagem
mkdir new_root/old_root
# execute o pivot_root trocando os mount points raiz atual para old_root e movendo o raiz para new_root
cd new_root
pivot_root . old_root
# Verifique que o / atual é o montado em rootfs
ls /tmp/rootfs.txt
# monte o /proc no novo namespace
mount -t proc none /proc/
# verifique que os pontos de montagem do sistema host ainda estão disponíveis
mount
# Remova a referência para os mount points do host
umount -l /old_root
# verifique o atual mtab
mount
```

lab 2

Objetivo: Testar os namespaces de PID REDE e IPC

Continuando do anterior

1. Execute o unshare criando uma nova namespace de PID

```
 unshare -pf --mount-proc=/proc /bin/bash
# execute o ps -ef para verificar a nova arvore de processos
ps -ef
```

2. Execute o unshare criando um namespace de rede

```
 # Listando a tabela de rotas do host
ip route list
# Listando as conexões Existentes
netstat -anp
# executando um ping para a internet
ping 8.8.8.8
# Crie o namespace de rede
unshare -npf --mount-proc=/proc /bin/bash
# Listando a tabela de rotas do contêiner
ip route list
# Listando as conexões exitestentes
netstat -anp
# executando um ping para a internet
ping 8.8.8.8
# Saindo do namespace
exit
```

3. Criando um recurso IPCS

```
 # Crie uma área de memória compartilhada
ipcmk --shmem 4096
# Liste os recursos
ipcs
# crie a namescpace de IPC
unshare -ipf --mount-proc=proc /bin/bash
# Liste os recursos na namespace
ipcs
# Saindo da namespace
```

lab 4

Objetivo: Demonstrar o uso da comunicação entre namespaces

Você precisará de dois terminais. O primeiro na namnespace de rede. o Segundo para executar os comandos

1. *No Terminal 2*, cCrie uma ancora de filesystem para suas namespaces.

```
# como root
 sudo su -
 mkdir -p /namespaces/001
 touch /namespaces/001/{net,ipc,mount,pid}
```

2. *No Terminal 1*, entre na namespace de rede isolada

```
 unshare --net=/namespaces/001 /bin/bash
 # verifique as interfaces disponíveis
 ip link ls
 # carregar a interface lo
 ip link set dev lo up
```

3. *No Terminal 2*, configurar a rede no namespace 001

```
 # Criar o par de vethX
ip link add veth0 type veth peer name veth1
# Atribuir a veth0 ao namespace
ip link set veth0 netns /root/namespaces/net
# Definir um ip para a rede
ip addr add 10.23.0.1/30 dev veth1
# Carregar a interface
ip link set dev veth1 up
```

4. *No Terminal 1*, configurar a rede

```
 # Atribuir ip à interface
ip addr add 10.23.0.2/30 dev veth0
# inicializar a interface
ip link set veth0 up
# Teste de rede
ping 10.23.0.1
```

lab 5

Objetivo: Demonstrar o uso de cgroups

1. Compilar o gastador de recursos

```
docker run --rm -it -v .:/build golang
cd src
go build -o waste-resource
```

2. Descubra a versão do cgroup em uso

```
$ mount |grep cgroup
 # Se o cgroup for v2, vc verá uma linha para o cgroup semelhante a abaixo
 cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,relatime,nsdelegate,memory_recursiveprot)
 # Se for um cgroup v1 vera uma linha para cada control group, semelhante a abaixo
 tmpfs on /sys/fs/cgroup type tmpfs (rw,nosuid,nodev,noexec,relatime,mode=755)
 cgroup2 on /sys/fs/cgroup/unified type cgroup2 (rw,nosuid,nodev,noexec,relatime)
 cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
 cgroup on /sys/fs/cgroup/cpu type cgroup (rw,nosuid,nodev,noexec,relatime,cpu)
 cgroup on /sys/fs/cgroup/cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpuacct)
 cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
 cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
 cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
 cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
 cgroup on /sys/fs/cgroup/net_cls type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls)
 cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
 cgroup on /sys/fs/cgroup/net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_prio)
 cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
 cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
 cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
 cgroup on /sys/fs/cgroup/misc type cgroup (rw,nosuid,nodev,noexec,relatime,misc)
```

3. Criando um grupo para acomodar a memória limitada
    1. Usando cgroup v1

```
 # Va para diretório do cgroup de memória
cd /sys/fs/cgroup/memory
# crie um subdiretorio para representar o subgrupo de memória
sudo mkdir grupo-0
# entre no novo grupo e veja os controles com ls
cd grupo-0
ls
# defina o limite de memória em 20 megas
echo $((20 * 1024 * 1024 ))| sudo tee memory.limit_in_bytes
# verifique o limite de memória
cat memory.limit_in_bytes
```

    2. Usando o cgroup v2

```
 # Va para diretório do cgroup raiz
cd /sys/fs/cgroup
# crie um subdiretorio para representar o novo subgrupo
sudo mkdir grupo-0
# entre no novo grupo e veja os controles com ls
cd grupo-0
ls
# defina o limite de memória em 20 megas
echo $((20 * 1024 * 1024 ))| sudo tee memory.max
# verifique o limite de memória
cat memory.max
```

4. Testando o limite de memória com o gastador de recursos
    1. Obtenha o PID do Terminal 1

- No terminal 1

```
 # vá para onde está o consumidor de memória
cd labX
 # crie um novo shell para não limitar o atual
 bash
 # obtenha o PID do shell utilizado
echo $$
```

2. No Teminal 2, adicione o PID do Terminal 1 no grupo-0
    1. Usando cgroup v1

    ```
     cd /sys/fs/cgroup/memory/grupo-0
    echo {PID} > tasks
    ```

    2. Usando cgroup v2

    ```
     cd /sys/fs/cgroup/grupo-0/
    echo {PID} > cgroup.procs
    ```

3. No Terminal 1. Teste o limite de memória. O $? mostra a saída do último comando utilizado. o 137 é saída por oom-kill

```
 ./waste-resource -memory-hog 1
 echo $?
 ./waste-resource -memroy-hog 3
 echo $?
 # saia do processo limitado
 exit
```

5. Criar um grupo com limite de cpu de 20% de 1 core. O cpu é limitado fracionando-o no tempo. Para limitar o grupo em 20%, daremos a ele 20 a cada 100 milisegundos de tempo de cpu. A unidade mínima de alocação que o cgroups permite e trabalha é o microsegundo. Isso significa que daremos 20000 a cada 100000 microsegundos de CPU.
    1. Utilizando o cgroups v1

    ```
     cd /sys/fs/cgroup/cpu
     sudo mkdir grupo-0
     cd grupo-0
     echo 100000 | sudo tee cpu.cfs_period_us
     echo 20000 | sudo tee cpu.cfs_quota_us
    ```

    2. Utilizando o cgroups v2, o grupo já foi criado no lab anterior, iremos somente definir o limite

    ```
     cd /sys/fs/cgroup/grupo-0
     echo "20000 100000" | sudo tee cpu.max
    ```

6. Testando o limite de CPU com o gastador de recursos
    1. Obtenha o PID do Terminal 1 No terminal 1

    ```
     # vá para onde está o consumidor de recursos
    cd labX
    # crie um novo shell para não limitar o atual
    bash
    # obtenha o PID do shell utilizado
    echo $$
    ```

    2. No Teminal 2, adicione o PID do Terminal 1 no grupo-0
        1. Usando cgroup v1

        ```
         cd /sys/fs/cgroup/cpu/grupo-0
        echo {PID} > tasks
        ```

        2. Usando cgroup v2

        ```
         cd /sys/fs/cgroup/grupo-0/
        echo {PID} > cgroup.procs
        ```

    3. No Terminal 1. Teste o limite de cpu

```
./waste-resource -waste-cpu 1 &
# Verifique o consumo com o comando top
top
# Saia do terminal limitado
exit
```

lab 6

Objetivo: Aprender os comandos básicos do docker logs

1. Execute um contêiner que gera logs

```
docker run --name lab6_busybox -d busybox sh -c 'while true; do echo "Log entry at $(date)"; sleep 5; done'
sleep 60
```

2. Ver os 5 ultimos logs

```
docker logs -n 5 lab6_busybox
```

3. Ver os logs dos ultimos 10s

```
docker logs --since 10s lab6_busybox
```

4. Ver os logs até 30s atrás

```
docker logs --until 30s lab6_busybox
```

5. Acompanhar os logs em tempo real

```
docker logs -f lab6_busybox
```

6. Encerrar

```
docker stop lab6_busybox
docker logs lab6_busybox
docker rm lab6_busybox
```

lab 7

Objetivo: Ver como um volume local pode ser utilizado para persistir dados

1. Crie um volume docker e monte-o em um contêiner

```
docker volume create lab7_volume_named
docker run --name lab7_container_1 --mount source=lab7_volume_named,target=/data -it  busybox
$ echo "Inside Container id $(hostname)" >/data/lab71.txt
$ exit
```

2. Crie um contêiner com o volume nomeado junto ao run

```
docker run --name lab7_container_2 --mount source=lab7_volume_named2,target=/data -it busybox
$ echo "Inside id $(hostname)"  > /data/lab71.txt
$ exit
```

```
docker run --name lab7_container_3 -v /data -it busybox
$ echo "Inside id $(hostname)"  > /data/lab71.txt
$ exit
```

3. Acesse os dados de outro contêiner

```
docker volume ls #verifique o nome dos conteienrs
docker run --name lab7_container_2 --mount source=lab7_volume_named,target=/data --mount source=lab7_volume_named2,target=/dat
$ echo "Inside Conatainer id $(hostname)" > /data4/lab71.txt
$ cat /data/lab71.txt
$ cat /data2/lab71.txt
$ cat /data3/lab71.txt
exit
```

4. Liste os volumes, os conteiners que os acessam e tente removê-los. Reflita o que aconteceu e remova-os conteiners

```
docker volume ls
docker rm <<volumes>>
```

lab 8

Objetivo: Entender como funcionam as redes bridge no Docker e como conectar contêineres usando essa rede padrão

1. Verificar Redes Existentes:
   - Liste as redes existentes no Docker.

```
docker network ls
```

2. Criar uma Rede Bridge:
   - Crie uma nova rede bridge chamada `minha_bridge`.

```
docker network create minha_bridge
```

3. Executar Contêineres na Rede Bridge:
   - Execute dois contêineres `busybox` na nova rede `minha_bridge` e inicie um shell interativo.

```
docker run -dit --name conteiner1 --network minha_bridge busybox
docker run -dit --name conteiner2 --network minha_bridge busybox
```

4. Testar Conectividade entre os Contêineres:
   - Acesse o primeiro contêiner e use o comando `ping` para verificar a conectividade com o segundo contêiner.

```
docker exec -it conteiner1 sh
ping conteiner2
```

5. Limpar o Ambiente:
   - Saia do contêiner e remova os contêineres e a rede criada.

```
docker stop conteiner1 conteiner2
docker rm conteiner1 conteiner2
docker network rm minha_bridge
```

Lab 9

Objetivo: Entender como funciona a rede host no Docker e como ela difere das outras redes

1. **Executar um Contêiner na Rede Host:**
   - Execute um contêiner `nginx` na rede host.

```
docker run -d --name nginx_host --network host nginx
```

2. **Verificar a Conectividade:**
   - Acesse a aplicação Nginx através do IP do host na porta 80.

```
curl http://localhost
```

3. **Comparar com a Rede Bridge:**
   - Execute um contêiner `nginx` na rede bridge e acesse através da porta mapeada.

```
docker run -d --name nginx_bridge -p 8080:80 nginx
curl http://localhost:8080
```

4. **Limpar o Ambiente:**
    - Remova os contêineres criados.

```
docker stop nginx_host nginx_bridge
docker rm nginx_host nginx_bridge
```

Lab 10

Objetivo: Entender como utilizar um repositório docker, executando um local

1. Utilize o compose do lab10 da Unidade 2
    - Avalie como foi construído o compose

```
cd lab10
docker compose up -d
```

2. Baixe a imagem do bash para sua máquina e envie para seu repositório local

```
docker pull bash:latest
docker tag bash:latest localhost:5000/bash:latest
docker push localhost:5000/bash:latest
```

3. Navegue no diretório data mapeado para o repositório e procure entender a estrutura. Note que não existe um arquivo com a imagem bash mas uma coleção de arquivos com manifestos e imagens

Lab 11

Objetivo: Entender como gerenciar usuários e grupos dentro dos contêineres Docker para melhorar a segurança.

1. **Executar um Contêiner com Usuário Padrão:**
    - Execute um contêiner `ubuntu` e verifique o usuário atual.

```
docker run -it --name user_test ubuntu
whoami
```

2. **Adicionar um Novo Usuário:**
    - Dentro do contêiner, adicione um novo usuário chamado `novo_usuario`.

```
apt-get update
apt-get install -y sudo
adduser novo_usuario
usermod -aG sudo novo_usuario
```

3. **Executar o Contêiner com o Novo Usuário:**
    - Saia do contêiner e execute-o novamente com o novo usuário `novo_usuario`.

```
docker commit user_test user_test_image
docker rm user_test
docker run -it --name user_test --user novo_usuario user_test_image
whoami
```

4. **Limpar o Ambiente:**
    - Remova o contêiner e a imagem criados.

```bash docker rm user_test docker rmi user_test_image

# Lab 12

### Objetivo: Entender como limitar os recursos (CPU e memória) dos contêineres Docker para melhorar a segurança e a estabilidade do sistema.

1. **Crie uma imagem com o gastador de recursos do lab5
```

```
cd lab5
docker build -t waste-resources .
```

2. **Executar um Contêiner com Limites de CPU e Memória:**

```
docker run -d --name limited_container --cpus=".5" --memory="256m" waste-resources -waste-cpu 1
docker run -d --name limited_container --cpus=".5" --memory="256m" waste-resources -hog-memory 25
```

3. **Verificar Limites de Recursos:**
   - Use o comando `docker stats` para verificar os recursos consumidos pelo contêiner.

```
docker stats limited_container
```

4. **Limpar o Ambiente:**
   - Remova o contêiner criado.

```
docker stop limited_container
docker rm limited_container
```

Laboratório 13

Objetivo: Aprender a aplicar políticas de segurança utilizando `--cap-drop` para remover capacidades específicas em contêineres Docker

1. **Executar um Contêiner sem Remover Capacidades:**
   - Execute um contêiner `busybox` e teste a capacidade de usar `ping`, DNS e TCP.

```
docker run --rm -it busybox
```

   - Dentro do contêiner, teste os seguintes comandos:

```
ping -c 2 google.com
nslookup google.com
wget http://example.com
exit
```

2. **Executar um Contêiner com `--cap-drop` para Remover Capacidades:**
   - Execute um contêiner `busybox` removendo a capacidade `CAP_NET_RAW` que é necessária para o comando `ping`.

```
docker run --rm -it --cap-drop=NET_RAW busybox
```

3. **Testar as Restrições:**
   - Dentro do contêiner, tente usar os mesmos comandos novamente:

```
ping -c 2 google.com    # Deve falhar
nslookup google.com     # Deve funcionar
wget http://example.com # Deve funcionar
```

4. **Analisar o Comportamento:**
   - Observe que o comando `ping` falha enquanto os comandos `nslookup` e `wget` ainda funcionam, indicando que as capacidades de DNS e TCP estão intactas.