



AR_baby-groot.jpg

Conteneur "à la roots"

Prérequis

Votre environnement doit être prêt : voir [Activity - Préparer son environnement Linux](#). Toutes les commandes se tapent **dans votre VM Linux**, jamais dans PowerShell ni dans le Terminal de macOS.

Sous Lima (macOS)

Après `limactl shell tp`, faites toujours `cd ~` avant de commencer. Vous arrivez sinon dans les fichiers de votre Mac, montés en lecture seule — et le TP ne peut pas y fonctionner. Un `pwd` doit afficher un chemin sous `/home/`, jamais sous `/Users/`.

Si vous êtes déjà root

C'est le cas sur Killercoda. Les `sudo` du TP sont alors inutiles, mais inoffensifs : laissez-les. Si la commande `sudo` n'existe pas chez vous, retirez simplement le préfixe.

Ce qu'on cherche à obtenir

À la fin de ce TP, vous aurez lancé un shell qui :

- voit un autre système de fichiers que le vôtre ;
- a son propre nom de machine ;
- se croit le premier processus de la machine ;
- n'a plus accès au réseau ;
- tourne en `root` sans en avoir tous les pouvoirs.

Autrement dit : un conteneur. Sans Docker, et sans rien installer de plus.

L'objectif n'est pas de retenir les commandes

`unshare` et `capsh` ne sont pas des outils que vous utiliserez au quotidien. Ce qui compte, c'est de **voir** ce que chaque brique change. Notez vos observations au fil du TP et gardez-les dans un coin : on en aura besoin au débrief.

Partie 1 — La situation « avant »

Avant d'isoler quoi que ce soit, regardez où vous êtes. Exécutez :

```
hostname
id
ps -e | wc -l
ip -brief link
ls /
```

Notez vos réponses, on les comparera à la fin :

- Quel est le nom de votre machine ?
- Sous quelle identité travaillez-vous ?
- Combien de processus voyez-vous ?
- Combien d'interfaces réseau ?

Partie 2 — `chroot` : changer de système de fichiers

Déballez l'archive Alpine que vous avez téléchargée :

```
cd ~/tp-conteneur
mkdir -p rootfs
tar -xzf alpine-minirootfs-*.tar.gz -C rootfs
ls rootfs
```

L'option `-f` se place en dernier

Dans un groupe d'options courtes collées, celle qui attend un argument doit être la dernière. `tar -xzf archive` fonctionne, `tar -xzf archive` échoue : `-f` recevrait `z` comme nom de fichier.

Vous avez maintenant une arborescence Linux complète dans `rootfs`. Entrez dedans :

```
sudo chroot rootfs /bin/sh
```

Puis, à l'intérieur :

```
ls /  
cat /etc/os-release  
hostname  
ps -e
```

Trois questions sur ce que vous venez d'observer :

1. Le contenu de `/` a-t-il changé ? Et `/etc/os-release` : sur quelle distribution êtes-vous ?
2. `hostname` affiche-t-il la même chose qu'en Partie 1 ?
3. `ps -e` fonctionne-t-il ? Pourquoi, à votre avis, n'affiche-t-il rien ici alors qu'il fonctionnait avant ?

Sortez avec `exit`.

`ps` ne voit rien... ou voit tout ?

`ps` lit ses informations dans `/proc`, qui n'est pas monté dans notre `rootfs`. Montons-le pour voir :

```
sudo mkdir -p rootfs/proc  
sudo mount --bind /proc rootfs/proc  
sudo chroot rootfs /bin/sh
```

À l'intérieur :

```
ps -e | wc -l  
ip -brief link
```

- Combien de processus voyez-vous maintenant ?
- Est-ce le nombre d'un système isolé, ou celui de la Partie 1 ?

Sortez (`exit`) et démontez :

```
sudo umount rootfs/proc
```

Première conclusion

`chroot` change le **système de fichiers, et rien d'autre**. Les processus, le nom de machine, le réseau : tout est encore celui de l'hôte. Un `chroot` n'est pas un conteneur.

Partie 3 — Les namespaces, un par un

C'est `unshare` qui crée les espaces de noms. Commencez par regarder ce qu'il sait faire :

```
unshare --help
```

Repérez les options qui correspondent aux mots `uts`, `pid`, `net` et `mount`. Elles vont servir une par une.

a) `uts` — son propre nom de machine

```
sudo unshare --uts /bin/bash
hostname pseudo-conteneur
hostname
exit
hostname
```

- Que montre `hostname` avant le `exit` ? Et après ?
- Qu'en déduisez-vous sur la portée du changement de nom ?

b) `pid` — devenir le premier processus

```
sudo unshare --pid --fork --mount-proc /bin/bash
ps -e
echo "mon PID : $$"
exit
```

⚠ Deux options obligatoires, non devinables

`--fork` est indispensable avec `--pid` : sans elle, `unshare` lui-même resterait dans l'ancien espace et le shell ne serait pas le premier processus. `--mount-proc` remonte `/proc` pour que `ps` lise le **nouvel** espace et non celui de l'hôte. Retirez l'une ou l'autre et relancez pour voir ce qui change.

- Combien de processus `ps -e` affiche-t-il ?
- Quel est le PID de votre shell ?
- Les processus de l'hôte ont-ils disparu, ou sont-ils seulement invisibles ?

c) `net` — plus de réseau

```
sudo unshare --net /bin/bash
ip -brief link
ping -c1 1.1.1.1
exit
```

- Quelles interfaces restent ?
- Que répond `ping` ? Le réseau est-il cassé, ou simplement absent de cet espace ?

d) Tout ensemble, plus le `chroot`

Une seule commande pour empiler les quatre briques :

```
sudo unshare --uts --pid --fork --net --mount chroot ~/tp-conteneur/rootfs /bin/sh
```

À l'intérieur, montez `/proc` puis observez :

```
mount -t proc proc /proc
hostname pseudo-conteneur
hostname
ps -e
ip -brief link
ls /
cat /etc/os-release
```

Vous êtes dans votre conteneur. Sortez avec `exit` .

Partie 4 — Capabilities : root, mais pas tout-puissant

Dans le conteneur, vous êtes `root` . Mais `root` n'est pas un bloc : c'est une collection de pouvoirs séparés, les *capabilities*.

Comparez les capacités de `root` et celles d'un utilisateur ordinaire. Les deux commandes passent par `sudo` , pour donner le même résultat que vous ayez démarré en `root` ou non :

```
sudo capsh --print | grep -i current
sudo -u nobody capsh --print | grep -i current
```

- Combien de capacités voyez-vous dans chaque cas ?
- La seconde ligne est-elle vide, ou presque ?

 `sudo -u nobody`

Exécute la commande sous l'identité de `nobody` , un compte sans aucun privilège. C'est ce qui nous permet de voir les deux extrêmes côte à côte, quelle que soit l'identité sous laquelle vous avez ouvert votre terminal.

Maintenant, retirons des capacités à `root` et regardons ce qui casse. Préparez un fichier que seul son propriétaire peut lire :

```
sudo sh -c 'echo "code du coffre : 1234" > /tmp/secret'
sudo chmod 600 /tmp/secret
sudo chown nobody /tmp/secret
```

Le fichier appartient à `nobody` et n'est lisible que par son propriétaire. Pourtant `root` le lit sans difficulté :

```
sudo cat /tmp/secret
```

Retirez-lui la capacité qui permet d'ignorer les permissions de fichiers, et refaites exactement la même lecture :

```
sudo capsh --drop=cap_dac_override -- -c 'cat /tmp/secret'
```

Le code du coffre s'affiche encore. Essayez maintenant en retirant une capacité de plus :

```
sudo capsh --drop=cap_dac_override,cap_dac_read_search -- -c 'cat /tmp/secret'
```

- Cette fois, que répond `cat` ?
- Vous êtes toujours `root` dans les trois cas. Qu'est-ce qui change, alors ?
- Pourquoi la première tentative ne suffisait-elle pas ?

🔗 Deux capacités pour un même pouvoir

`cap_dac_override` ignore **toutes** les vérifications de permissions, en lecture comme en écriture. `cap_dac_read_search` n'en ignore qu'une partie : la lecture des fichiers et le parcours des répertoires. Pour une simple lecture, chacune des deux suffit — il faut donc retirer les deux.

C'est la leçon la plus utile de cette partie : restreindre un conteneur suppose de savoir **ce que couvre exactement** chaque capacité. En oublier une voisine, et la restriction ne sert à rien.

Un conteneur tourne presque toujours en `root`, mais avec une liste de capacités volontairement réduite. « Être root dans un conteneur » n'est donc pas « être root sur la machine ».

Pour finir

Vous venez de construire, à la main, ce que `docker run` fait en une commande. Gardez vos observations sous les yeux : la théorie qui suit ne fait que nommer ce que vous avez vu.