Conteneurs

1. Préambule

L'objectif de TP est d'explorer les mécanismes techniques sur lesquels reposent les technologies de conteneurs, notamment Docker.

2. Prérequis

Une machine virtuelle au format VirtualBox est utilisée pour ce TP :

- SSE_TP2_Debian : utilisation d'une distribution Debian Linux 9 installée dans une partition chiffrée (mot de passe *tp*) et dotée d'un compte utilisateur nommé tp (mot de passe *tp*) ayant l'autorisation de lancer des commandes à l'aide de sudo.

Par ailleurs, une image ISO contenant les différents fichiers sources en C est disponible sur Moodle.

En raison de l'installation de Docker et de paquets supplémentaires sous Debian, la VM Debian doit avoir accès à Internet.

3. Mécanismes techniques

Exercice 3-1[1]

Objectif : Connaître les limites de chroot

Les limites de chroot sont généralement bien connues. Si vous les connaissez déjà, autant passer à l'exercice 3-2.

Sur la VM Debian, ouvrir une invite de commande avec le compte tp et taper les commandes suivantes pour créer une arborescence destinée à constituer une nouvelle racine :

```
mkdir /tmp/test_chroot
mkdir /tmp/test_chroot/bin
cp /bin/bash /tmp/test_chroot/bin
chroot /tmp/test_chroot
sudo chroot /tmp/test_chroot
```

Pourquoi est-ce que le premier appel à la commande chroot (celui sans sudo) ne fonctionne pas? La deuxième appel fonctionne ? Pourquoi ?

_____

_____

_____

1 Inspiré par https://gist.github.com/FiloSottile/6976188

Remarquer les dépendances de bash, les copier dans l'arborescence créée puis tester que bash fonctionne correctement :

```
ldd /bin/bash

mkdir /tmp/test_chroot/lib /tmp/test_chroot/lib64
cp /lib/x86_64-linux-gnu/libtinfo.so.5 /tmp/test_chroot/lib
cp /lib/x86_64-linux-gnu/libdl.so.2 /tmp/test_chroot/lib
cp /lib64/ld-linux-x86-64.so.2 /tmp/test_chroot/lib64/
cp /lib/x86_64-linux-gnu/libc.so.6 /tmp/test_chroot/lib
sudo chroot /tmp/test_chroot
bash-4.4# exit
```

Il est possible de réaliser la même opération avec /bin/ls pour afficher le contenu de la nouvelle arborescence.

Générer le programme permettant de s'échapper de la nouvelle arborescence racine.

```
gcc -static -o unchroot unchroot.c

mkdir /tmp/test_chroot/bin
cp /home/tp/unchroot /tmp/test_chroot/bin
sudo chroot /tmp/test_chroot
bash-4.4# unchroot
# ls -l
```

Quelle différence peut-on faire entre l'appel à exit et l'utilisation du programme unchroot ? Quels sont les privilèges nécessaires pour faire fonctionner le programme unchroot ?

Au final, la commande chroot permet-elle une bonne isolation au niveau système de fichiers ? Que pourrait-on envisager de faire pour l'améliorer ?

Exercice 3-2

Objectif : appréhender les concepts de *namespace*

Les *namespace* du noyau Linux peuvent être manipulés de différentes façon, notamment au moyen de l'appel système clone(). Pour un soucis de simplicité et pour éviter de faire appel à trop de code source, des outils en ligne de commande seront utilisés.

Dans un premier shell, exécuter les commandes suivantes pour créer un processus bash situé dans un nouveau *namespace* de type PID :

```
sudo unshare --fork --pid --mount-proc bash
top
```

Dans un autre shell, exécuter les commandes suivantes :

```
ps -up 1
ps aux | grep top
```

Que remarquez-vous concernant les processus et leur PID ? Que pouvez-vous conclure sur le rôle du *namespace* PID ?

Identifier le PID, vu du système debian, du processus bash qui a lancé la commande top dans le premier shell. Une fois le PID identifié, taper la commande suivante dans le premier shell :

```
ls -l /proc/$$/ns
```

puis les commandes suivantes dans le second shell (en remplaçant 1015 par le PID précédemment identifié) :

```
ls -l /proc/1015/ns
ls -l /proc/$$/ns
```

Que constatez-vous en examinant les différences ?

Dans un premier shell, exécuter les commandes suivantes pour sortir du *namespace* PID créé précédemment et en créer un nouveau en enlevant l'option --mount-proc :

```
exit
sudo unshare --fork --pid bash
echo $$
ps -up 1
ls -l /proc/$$/ns
```

Quel résultat obtenez-vous ?

Fermer les différents shell ouverts avec la commande unshare et comparer le contenu de /proc et dans les 2 cas suivants :

```
sudo unshare --fork --mount-proc --pid bash
sudo unshare --fork --pid bash
```

Pouvez-vous émettre une hypothèse qui explique la différence ?

Exercice 3-3

Objectif : manipuler les mécanismes de *cgroup*

SystemD utilise les *cgroup* avec des noms du type **user.slice** ou **system.slice**. Il apporte des commandes comme `systemd-cgls` et systemd-cgtop qui permettent respectivement de lister les différents processus rattachés à un *cgroup* pour chaque contrôleur et d'afficher la consommation de ressource par *cgroup.*

Afficher les différents types de contrôleurs de ressources (`man cgroups` pour une description du rôle de chaque contrôleur) présents sur le système, puis ceux qui impactent le shell courant. Le shell courant est-il suivi par tous les contrôleurs présents ? Pour chaque contrôleur, est-il rattaché à un *cgroup* mis en place par SystemD ou le *cgroup* par défaut (*cgroup* qui n'a pas de nom) ?

```
mount -t cgroup
ls /sys/fs/cgroup
cat /proc/$$/cgroup
```

Créer un nouveau cgroup pour le contrôleur pids, lui affecter le shell courant, s'assurer que l'affectation est effective et afficher les tâches affectées au *cgroup* créé à l'aide des commandes suivantes :

```
sudo su
```

```
cd /sys/fs/cgroup/pids/
mkdir tp
echo "$$" > tp/tasks
grep pids /proc/$$/cgroup
cat tp/tasks
```

Combien de PID sont affichés par la dernière commande. Comment l'expliquez-vous ?

Dans le cas du contrôleur *pids*, le fichier pids.events contient le nombre de tentatives de dépassement des limites imposées pour un *cgroup* donné.

```
cat tp/pids.events
echo 2 > tp/pids.max
sleep 20s &
cat tp/pids.current
```

Quel résultat obtient-on ici ? La valeur contenu dans le fichier pids.event a-t-elle évoluée ?

Pour vérifier l'efficacité du mécanisme, il est possible de tester une attaque en déni de service comme une *fork bomb* avec une limite (par exemple, 100) et sans limite (« max ») :

```
echo 100 > tp/pids.max
echo "max" > tp/pids.max
```

Il est conseillé de sauvegarder les données utiles qui se trouveraient dans la machine debian (et potentiellement dans la machine physique si Virtual Box ne faisait pas correctement la gestion de l'allocation du processeur) avant de lancer ce test. Des exemples de *fork bomb* sont simples à trouver, par exemple :

https://www.admin-linux.fr/fork-bombe-en-bash-comprendre-et-sen-proteger/

Quel est le comportement de la machine dans les 2 cas ?

D'une manière similaire à ce qui a été fait pour le contrôleur *pids*, mettre en place un cgroup tp pour le contrôleur *memory*.

La documentation de référence sur ce contrôleur se trouve à l'adresse (d'autres références[2] plus « colorées » peuvent être néanmoins utilisées) : https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt

Mettre en place une limite à 100Mo de mémoire utilisateur (par opposition à la mémoire noyau) sans se soucier du swap. Pour tester cette limite, plusieurs options peuvent être considérées, par exemple :

- installer les outils stress ou stress-ng (à partir des paquets debian éponymes)
- écrire un programme en C qui alloue une quantité de mémoire passée en paramètre

Quel « fichier » de configuration faut-il utiliser pour mettre en place la limite fixée ?

4. Docker

Installation de Docker sous Debian

Suivre les consignes données sur la page https://docs.docker.com/install/linux/docker-

2 Par exemple, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-memory

ce/debian/#install-using-the-repository

Exercice 4-1

Objectif : découvrir les certains mécanismes techniques à base du fonctionnement de Docker

Afin de distinguer les commandes à taper dans la machine debian de celles à taper dans le conteneur, les commandes seront préfixées du prompt idoine.

Pour cet exercice, un conteneur reposant sur bash sera utilisé.

```
tp@ses-tp-debian:~$ sudo docker pull bash
tp@ses-tp-debian:~$ sudo docker run --rm -it bash
bash-5.0# sleep 60s
```

A l'aide du processus sleep lancé dans le conteneur, identifier l'arborescence des processus parents du conteneur bash puis noter le PID de chaque processus et le nom de l'utilisateur ayant lancé chaque processus.

A l'aide des commandes courantes (top, free, ps, …), identifier depuis le conteneur la quantité de mémoire exposée dans le conteneur ainsi que les processus présents et leur PID. Que remarque-t-on ? Est-il possible de formuler une hypothèse sur la gestion des PID ?

Utiliser les commandes suivantes pour vérifier l'usage de *namespace* sur le système (remplacer 1114 par le PID du processus bash du conteneur) :

```
tp@ses-tp-debian:~$ sudo lsns
```

```
tp@ses-tp-debian:~$ sudo ls -l /proc/1114/ns
```

Quels sont les types de *namespace* mis en place spécifiquement pour le conteneur *bash* ? D'où proviennent les *namespace* qui ne sont pas mis en place spécifiquement pour le conteneur ?

Lancer un deuxième conteneur bash avec la commande suivante et comparer la gestion des *namespace* entre les 2 conteneurs.

```
tp@ses-tp-debian:~$ sudo docker run --rm --pid="host" -it bash
```

Cela confirme-t-il l'hypothèse émise précédemment ? S'agit-il d'une bonne pratique en terme de sécurité ?

Cf. https://docs.docker.com/engine/reference/run/#pid-settings---pid pour plus de détails sur le paramètres --pid.

Arrêter le 2è conteneur (à l'issue, la commande **`sudo docker ps`** ne doit afficher qu'un seul conteneur) :

```
bash-5.0# exit
```

Identifier si docker utilise les cgroups pour le conteneur bash et comparer la vision des *cgroup* depuis la machine debian et le conteneur bash (remplacer 1114 par le PID du processus bash du conteneur) à partir des informations disponibles pour les processus :

```
tp@ses-tp-debian:~$ cat /proc/1114/cgroup
bash-5.0# cat /proc/$$/cgroup
```

Démarrer un autre conteneur

```
tp@ses-tp-debian:~$ sudo docker run --rm --pids-limit=3 -m 100M -it bash
```

Comparer la valeur de **/sys/fs/cgroup/memory/memory.limit_in_bytes** pour les 2 conteneurs, puis la quantité de mémoire indiquée par la commande **free**. Sont-elles identiques pour les 2 conteneurs ? Que peut-on en conclure ?

Comparer la valeur de **/sys/fs/cgroup/pids/pids.max** pour les 2 conteneurs. Est-elle identique pour les 2 conteneurs ? Le premier conteneur est-il sensible a un type d'attaque en particulier ?

Exercice 4-2

Objectif : examiner l'utilisation des capacités par le moteur runC de Docker

Dans les versions récentes de Docker CE, seul le moteur runC est disponible (le support de LXC a été abandonné dans la version 1.10). Pour vérifier le moteur utilisé, il faut taper la commande suivante :

```
tp@ses-tp-debian:~$ sudo docker info
```

Pour étudier les capacités, il est possible d'utiliser les techniques vues dans les précédents TP ou d'utiliser la commande pscap liste les capacités des processus. Compte-tenu des nombreux processus à suivre dans cet exercice, l'utilisation de la commande pscap sera privilégiée.

```
tp@ses-tp-debian:~$ sudo apt-get install libcap-ng-utils
```

Comparer et commenter les capacités attribuées au processus sleep du conteneur bash dans les 4 cas suivants. Quelle recommandation pourrait-on émettre sur la gestion des capacités dans les

conteneurs ?

```
tp@ses-tp-debian:~$ sudo docker run --rm -d bash sleep 5s > /dev/null; pscap | grep sleep
tp@ses-tp-debian:~$ sudo docker run --rm -d --privileged bash sleep 5s > /dev/null; pscap | grep sleep
tp@ses-tp-debian:~$ sudo docker run --rm -d --cap-drop=all bash sleep 5s > /dev/null; pscap
tp@ses-tp-debian:~$ sudo docker run --rm -d --cap-drop=all --cap-add=net_bind_service bash sleep 5s > /dev/null; pscap | grep sleep
```