Eléments de sécurité sur Docker

Ce document a pour but d'analyser les éléments de sécurité liés à Docker afin de mieux comprendre quels peuvent être les limites et les pièges à éviter. Nous allons dans un premier temps explorer les composantes de Docker vues sous l'angle de la sécurité puis décrire les risques liés aux containers Docker . Pour finir nous chercherons à utiliser les mécanismes existants de Docker afin de renforcer la sécurité de nos containers.

Briques de bases de Docker vues sous l'aspect sécurité.

Les containers partagent le Kernel de la machine hôte. Le Kernel va:

- Isoler efficacement des processus ou des groupes de processus les uns des autres.
- Contrôler la consommation des ressources de l'hôte par les containers.
- Donner les droits nécessaires et suffisants au container afin de rendre le micro-service pour lequel il a été instancié.

Comme LXC, la solution Docker est basée sur les composantes du Kernel que sont CGROUPS, les CAPABILITIES et les NAMESPACES. La combinaison de ces trois éléments donne l'illusion à un utilisateur d'être sur une entité serveur à part entière. Mais ce n'est pas le cas: un container est un "Canada dry". Cet aspect est moins marqué chez Docker qui **ne cherche pas** à mimer un vrai serveur avec l'ensemble de ses processus (logs , cron ...).

A l'aide de ce tryptique , le Kernel pourra éviter qu'un process ne puisse lire des fichiers dans une autre arborescence que la sienne. Il bâtit une barrière logique qui va limiter le container: C'est un "chroot" (change root) "on steroid".

Le kernel pourra aussi donner aux containers les ressources nécessaires (IO, CPU, MEM ...) au travers des **CGROUPS**. Ils permettent d'apparier et de contrôler des groupes de processus. Ce sont les "chiens de bergers" des containers. Ils peuvent contingenter et isoler les processus afin d'éviter qu'ils ne s'approprient toutes les ressources aux dépens des autres containers ou de l'hôte.

Les **CAPABILITIES** Linux permettent de limiter les droits d'un utilisateur. C'est en particulier vrai pour le compte root.

Les **NAMESPACES** permettent de créer des environnements à la demande et de berner les processus qui ne voient que ce que l'on veut bien leurs faire voir: un réseau, un point de montage, des utilisateurs ...".

Décrivons maintenant ces trois éléments du Kernel.

Les NameSpaces

Implémentés au fur et à mesure des années dans le Kernel Linux on peut distinguer:

- Les Users namespaces qui permettent à un container d'avoir ses propres utilisateurs indépendamment des autres containers et de l'hôte. En particulier L'UID 0 sur un container peut être mappé sur un UID "lambda" de l'hôte.
- Les Network namespaces qui dotent chaque container de son propre environnement réseau.
- Les Mounts namespaces qui autorisent chaque container à avoir ses propres points de montage.

- Les PID namespaces qui permettent à chaque container d'avoir sa propre arborescence de processus. Deux processus différents peuvent avoir le même PID dans deux containers différents.
- Les IPC namespaces qui ajoutent l'échange inter-processus au sein du même container.
- Les UTS namespaces car il est essentiel que chaque container ait son propre nom.

Appels système liés aux namespaces

Trois appels système sont utilisés par les containers:

- **clone()**: Il crée un nouveau processus et un nouveau namespace auquel le processus est rattaché.
- **unshare()**: Il crée un nouveau namespace auquel on rattache un processus existant.
- **setns()**: Il permet à un processus de se rattacher à un namespace existant ou de s'en détacher.

Exemple : Création de deux Network NamesSpace:

```
# Les network NAMESPACES permettent
# de créer un environnement virtuel réseau pour le container.
ip netns add netns1 # Création du premier namespace
ip netns add netns2 # Création du second namespace
ip netns list # Affichage des namespaces créés
ip netns exec netns1 /bin/bash # On entre dans le premier namespace
ip a # On affiche les cartes réseaux visibles
```

Création de deux cartes réseaux virtuelles et ajout d'une IP

```
ip link add name vethnetns type veth peer name vethnetns-peer
ip link set vethnetns netns netns1
ip netns exec netns1 /bin/bash
ip addr add 192.168.1.105/24  dev vethnetns
ip link set vethnetns up
ip a
```

Ping entre deux Veth

```
ip netns exec netns2 /bin/bash
ifconfig vethnetns-peer 192.168.1.104 netmask 255.255.255.0 broadcast
192.168.1.255
ping 192.168.1.105
```

Les Capabilities :** "Sans les anneaux de pouvoir, Sauron n'est rien , sans les capabilities, root n'est rien".**

Les capabilities permettent de contrôler ce qu'un utilisateur est capable de faire. Le compte "root" dispose de toutes les capabilities et en particulier de la capabilitie CAP-SYS-ADMIN qui lui donne ses droits étendus.

Une capabilitie doit être sans hésiter supprimée si elle n'est pas utilisée par le container. Docker dote par défaut ses containers des capacités suivantes:

- CAP_CHOWN: Avoir la capacité de changer le propriétaire d'un fichier;
- CAP_DAC_OVERRIDE: Passer outre le contrôle d'accès (Posix ACL);
- CAP_FSETID: Avoir la capacité d'utiliser chmod sans limitation;
- CAP_FOWNER: Outrepasser le besoin d'être propriétaire du fichier;
- CAP_MKNOD: Avoir la capacité d'utiliser des fichiers spéciaux;
- CAP*NET*RAW: Avoir la capacité d'utiliser les sockets raw et packet (snifiing, binding);
- CAP_SETGID: Avoir la capacité de changer le GID;
- CAP_SETUID: Avoir la capacité de changer l'UID;
- CAP_SETFCAP: Avoir la capacité de modifier les capacités d'un fichier;
- CAP_SETPCAP: Avoir la capacité de modifier les capacités d'un autre processus;
- CAP*NET*BIND_SERVICE:, Avoir la capacité d'écouter sur un port inférieur à 1024;
- CAP*SYS*CHROOT: Avoir la capacité de faire un change root;
- CAP_KILL: Avoir la capacité de killer un processus;
- CAP*AUDIT*WRITE: Avoir la capacité d'écrire des logs Kernels (par exemple pour changer un password);

  L'ajout d'une capabilitie ajoute des vulnérabilités ce qui peut rompre l'isolation du container.

Comment visualiser les capabilities d'un processus sshd à l'aide de la commande getpcaps ?

```
ps -ef | grep ssh
pouchou    1967   1946  0 16:28 ?        00:00:00 /usr/bin/ssh-agent
/usr/bin/dbus-launch
rootcont   2323   2290  0 16:30 ?        00:00:00 /usr/sbin/sshd -D
root       2359   2199  0 16:31 pts/0    00:00:00 grep ssh
getpcaps 2323 # Récupération des capabilities via la commande getpcaps
Capabilities for `2323': =
cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,

cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,

cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,
                                        cap_setfcap+eip
```

Visualisation du chroot: position d'un fichier de mon container dans l'arborescence de l'hôte.

```
docker run -it debian 'touch locateme' # création d'un fichier locateme
# Le container a sa propre arborescence
ls
bin  boot  dev etc  home  lib lib64  media  mnt  opt proc  root  run  sbin
srv  sys  tmp  usr  var
touch locateme
```

```


# On localise le fichier créé dans le container sur l'hôte
updatedb
locate locateme
```

```
/var/lib/docker/aufs/diff/3a8f388658b99867fad64a6516c9c61208b23f36a8ec042ef
81c6141337037b0/locateme
```

Docker "chroote" ses containers et le fichier locateme se retrouve dans /var/lib/docker/aufs/diff/ID*du*container. L'identifiant du container est unique et généré lors de sa création. Docker fonctionne par défaut avec un système de fichier de type "Union File System". C'est un système de fichier en couches. Notre fichier "locateme" est stocké dans la couche en écriture de ce système de fichier.

Les risques identifiés

- La prise de contrôle du container : un processus d'un container est hacké via une faille de sécurité.L'absence de User namespace est un facteur aggravant d'autant plus que le processus du container appartient à root (car alorsroot container=root hôte). L'autre vulnérabilité peut être basée sur un surplus de "capabilities" du container permettant une escalade de privilèges, suivi de la compromission de l'hôte.
- L'épuisement des ressources du container et de l'hôte par déni de service d'un utilisateur local ("fork bomb" par exemple) ou à distance (Surcharges dues à des requêtes sur un serveur web ou l'ouverture abusive de connexions TCP.)
- L'attaque du "daemon Docker" par un utilisateur local à l'hôte. Un utilisateur appartenant au groupe docker doit être considéré comme ayant les droits root sur l'hôte. Un utilisateur du groupe docker peut accéder ainsi accéder à n'importe quel fichier sur l'hôte et les exécuter en tant que root.
- La récupération d'une Image docker sur le Docker Hub (présence possible d'une backdoor, d'une image basée sur un système d'exploitation obsolète et présentant des vulnérabilités.).
- La compromission de l'hôte et donc accès à l'ensemble des containers.

Vecteurs de renforcement de la sécurité

- La surface d'attaque offerte par les 340 appels systèmes présents dans le kernel Linux 4.1 est importante. Docker permet de filtrer les appels systèmes via seccomp-bpf. La documentation indique que la politique par défaut de Docker pour seccomp est de filtrer 40 appels systèmes.

- Apparmor/Selinux sont supportés par Docker ce qui permet de contrôler ce que peut faire un container (par exemple interdire au process Linux d'accéder à certains fichiers dans le container).

- Docker permet aussi de mettre en lecture seule le container tout en laissant certaines directories comme /tmp modifiables ce qui peut peut être utile par exemple pour des containers frontaux web qui n'ont pas de besoins en écriture sur le disque.

- Les images sur le docker hub peuvent être signées. On peut donc vérifier qu'elles n'ont pas été altérées lors du "docker pull". Docker fournit aussi un scanner de vulnérabilité si vous avez un compte payant.

Environnement des tests

Afin de pouvoir tester les dernières avancées de Docker en termes de sécurité il faut avoir un Kernel récent (4.5) et une version de Docker récente (1.11 à minima). Nous avons donc travaillé avec une Debian Jessie mais avec un Kernel backport et sans le système de fichier utilisé AUFS, utilisé par défaut par Docker (non intégré dans le Kernel backport).

Vous serez peut être amenés à faire un `sudo bash rm -rf /var/lib/docker/aufs` et à redémarrer Docker pour utiliser le système de fichier devicemapper en lieu et place d'AUFS.

Il est conseillé de travailler dans une machine virtuelle. (Aucun container n'a été maltraité durant l'écriture de ce document mais ce n'est pas le cas forcément des machines hôtes).

Si vous travaillez sur une machine virtuelle prévoyez au moins deux CPUs (pour voir la limitation CPU par CGROUP).

Installation de Docker et du kernel Backport

```
### Quick and dirty installation de Docker sous Linux
curl -fsSL https://get.docker.com/ | sh
```

```
sudo tee -a /etc/apt/sources.list <<< "deb
http://httpredir.debian.org/debian jessie-backports main"
sudo apt update && sudo apt install linux-image-amd64/jessie-backports
```

Limiter les pouvoirs de root dans un container ssh Docker via le contrôle des capabilities

On va générer une image avec un daemon ssh à l'intérieur de ce container et voir si on peut limiter les droits de root.

Création d'un container sshd

```
FROM debian:jessie
MAINTAINER Jmp <jean-marc.pouchoulon@iutbeziers.fr>
# On se sert du repository Debian de l'IUT
RUN echo "deb http://debian.iutbeziers.fr/debian/ jessie main"  \
> /etc/apt/sources.list
# il vaut mieux faire un seul apt-get afin de n'ajouter qu'un seul layer
# 1 RUN = 1 commande = 1 Layer
RUN apt-get update && apt-get install --no-install-recommends -y -q \
    openssh-server \
    git-core \
    vim  \
    sudo \
    man
# On installe notre microservice
# ici un serveur ssh.
RUN mkdir /var/run/sshd
RUN echo 'root:iutbeziers' | chpasswd
RUN sed -i 's/PermitRootLogin without-password/PermitRootLogin
        yes/'  /etc/ssh/sshd_config
# SSH login fix.
RUN sed 's@session\s*required\s*pam_loginuid.so@session \
       optional pam_loginuid.so@g' -i /etc/pam.d/sshd
# On permet au container de faire voir
```

```
# aux autres containers que son port 22 est ouvert
EXPOSE 22
# On lance le daemon ssh
CMD ["/usr/sbin/sshd", "-D"]
```

Construction de l'image du container ssh via un Docker Build:

```
cd buildssh
docker rmi jmp/sshd
docker build -t jmp/sshd .
```

On va maintenant construire notre container, en enlevant les capabilities une par une. Au final et de façon empirique on obtient un container à priori fonctionnel et plus sécurisé.

Voici le résultat obtenu (non testé en production).

```
bash docker run -d \
      --cap-drop=chown \
      --cap-drop=dac_override \
      --cap-drop=fowner \
      --cap-drop=fsetid \
      --cap-drop=kill \
      --cap-drop=setpcap \
      --cap-drop=mknod \
      --cap-drop=setfcap \
      --publish=2222:22 \
      --name serveurssh \
      --hostname serveurssh \
      jmp/sshd
```

Connexion au container:

```
sshpass -p 'iutbeziers' ssh -o StrictHostKeyChecking=no -p 2222
root@localhost 'uname -a;ps -ef;'
```

```
Linux serveurssh 4.5.0-0.bpo.2-amd64 #1 SMP Debian 4.5.3-2~bpo8+1 (2016-05-
13) x86_64 GNU/Linux
UID         PID   PPID  C STIME TTY          TIME CMD
root          1      0  0 13:33 ?        00:00:00 /usr/sbin/sshd -D
root         25      1  0 13:46 ?        00:00:00 sshd: root@notty
root         27     25  0 13:46 ?        00:00:00 bash -c uname -a;ps -ef;
root         29     27  0 13:46 ?        00:00:00 ps -ef
```

Utilisez les cgroups pour limiter les ressources.

Un attaquant peut saturer un container en lançant un déni de services. Si on ne limite pas les ressources prises par un container, c'est l'hôte du container qui sera en manque de ressources à son tour. Il est donc essentiel de limiter les ressources prises par un container via les CGROUPs.

On construit ici un container "stresseur" à partir de ce Dockerfile afin d'en limiter pour démonstration les ressources :

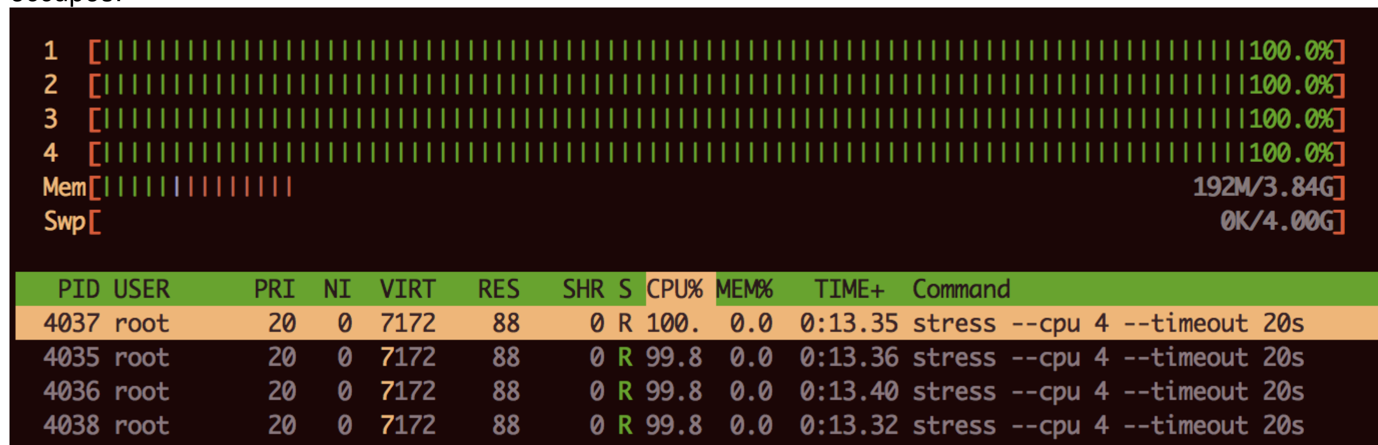
```
FROM debian:latest
RUN apt-get update && \
    apt-get install stress
```

```
cd ../buildstress
docker rmi jmp/stress
docker build -t jmp/stress .
```

Lancez le container "stresseur" et ouvrez une fenêtre htop pour voir la consommation de ressources sur l'hôte:

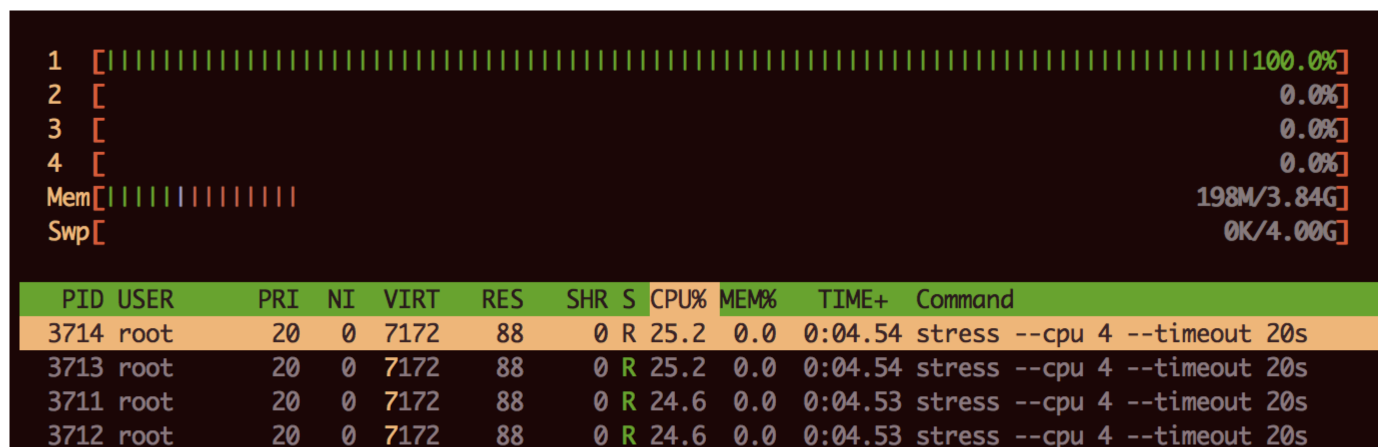
```
docker run -d  jmp/stress stress  --cpu 4 --timeout 20s
```

Vous devriez avoir le résultat suivant sur lequel on voit clairement que les quatre CPU sont occupés:



On ne permet maintenant au container de n'accéder qu'à un seul CPU de l'hôte pour limiter ses ressources.

```
docker run -it --cpuset-cpus="0"  --rm jmp/stress stress  --cpu 4 --timeout
20s
```



Pour entrainement refaites la même manipulation en stressant le container sur la mémoire.

Se protéger d'une fork bomb dans le container

Un forkbomb crée des processus qui vont eux-mêmes générer (appel système FORK) d'autres processus.

Cette commande permet de lister les ressources du container à l'aide de la commande docker stats.

```
docker stats --no-stream=True
```

```
CONTAINER           CPU %               MEM USAGE / LIMIT   MEM %
NET I/O             BLOCK I/O           PIDS
```

Dans une machine virtuelle lancez une "forkbomb" bash dans un container. Si tout se passe bien, votre container et votre machine virtuelle ne répondront plus, vous voilà prévenus...

```
docker exec -it deb1 /bin/bash -c ":(){ :|:& };:"
```

Recréez votre container en utilisant l'option ** --pids-limit ** et vérifiez que la nuisance de la forkbomb est limitée.

```
docker stop deb1 && docker rm deb1
docker run -d --name deb1 --hostname deb1 --pids-limit=20  debian /bin/sh -
c 'tailf /dev/null'
docker exec -it deb1 /bin/bash -c ":(){ :|:& };:"
```

Vérifier la signature d'une image Docker sur le Docker hub (nécessite d'avoir un compte sur Docker Hub)

Une fois votre clef privée générée pensez à la sauvegarder.

```
export DOCKER_CONTENT_TRUST=1 # le positionnement de cette variable permet
de signer nos containers
docker tag busybox pushou/dctrust:signed
docker push pushou/dctrust:signed
```

Utilisation de seccomp pour limiter l'utilisation de certains appels systèmes:

Vous devez vérifier que votre kernel supporte cette fonctionnalité: `bash cat /boot/config-`uname -r` | grep CONFIG_SECCOMP= CONFIG_SECCOMP=y` et que votre version de seccomp est supérieure à >= 2.2.1.
La fonctionnalité a uniquement été testée avec succès sur Ubuntu 16.
Il vous faut d'abord créer un fichier chmod.json qui va interdire le lancement de la commande chmod dans le container. `json { "defaultAction": "SCMP_ACT_ALLOW", "syscalls": [ { "name": "chmod", "action": "SCMP_ACT_ERRNO" } ] }`

```
docker run --rm -it --security-opt seccomp:chmod.json debian chmod 777
/etc/passwd
```

```
chmod: /etc/hostname: Operation not permitted
```

On peut trouver ici un profil par défaut qui désactive les appels systèmes non indispensables au fonctionnement du container.

Utilisation de APPARMOR afin de protéger un container obsolète de shellshock

Nous allons tout d'abord utiliser un container debian5 qui contient une version de bash vulnérable à shellshock. On construit l'image à partir de ce dockerfile:

```
FROM pblaszczyk/debian-lenny
MAINTAINER Jmp <jean-marc.pouchoulon@iutbeziers.fr>
RUN apt-get update && apt-get install --no-install-recommends -y -q \
apache2
ADD simple-cgi-bin.sh /usr/lib/cgi-bin/
RUN chmod +x /usr/lib/cgi-bin/simple-cgi-bin.sh
EXPOSE 80
CMD /usr/sbin/apache2ctl -D FOREGROUND
```

Le fichier simple-cgi-bin.sh contient:

```
!/bin/bash
echo "Content-type: text/plain"
echo echo echo "shellshockme if youcan"
```

Construction de l'image et création du container:

```
cd ../buildshellshock
docker build -t jmp/shellshockme .
docker run -d -p 80:80 --name=hitme --hostname=hitme jmp/shellshockme
```

```
hitme
hitme
Untagged: jmp/shellshockme:latest
Deleted:
sha256:0974b41738716b983acc85679eedb1d84a431d41f94ae1afa425cc1be2b98afd
Deleted:
sha256:7a9cb6b172f87b92720b9dbecf91447fa3e8853b89ad7da2b47ff21b6b6feb34
Deleted:
sha256:0b11f7a7f6affddc8d2ebf0e8beea622a01994e78c6de2f90972f81b106a7c1a
Deleted:
sha256:d40831b0d5ae7dcf5aa6d7a1fbe687c733b29168f3dfcf4d6a8b5ca4c71dad95
Deleted:
sha256:5bdb9d3c340f59fc74e3319f3eb7f2245cfae0a9ee40421fb67813fba9076090
Deleted:
sha256:0e4414265461b6ae7f3a1df79d4fade0b621ddf667cf18fbfb47ecf838b34816
```

```
Step 1 : FROM pblaszczyk/debian-lenny
 ---> 3c13189de511
Step 2 : MAINTAINER Jmp <jmp@iutbeziers.fr>
 ---> Using cache
 ---> 577efe4aef81
Step 3 : RUN apt-get update && apt-get install --no-install-recommends -y -
q apache2
 ---> Using cache
 ---> fade67c504a0
Step 4 : ADD simple-cgi-bin.sh /usr/lib/cgi-bin/
 ---> 0b4a7f8f774b
Removing intermediate container 4a29a14c4279
Step 5 : RUN chmod +x /usr/lib/cgi-bin/simple-cgi-bin.sh
 ---> Running in 15f9e9fa08bc
 ---> 456d030eb77e
Removing intermediate container 15f9e9fa08bc
Step 6 : EXPOSE 80
 ---> Running in fd58fc0cc35c
 ---> bd09f88d1659
Removing intermediate container fd58fc0cc35c
Step 7 : CMD /usr/sbin/apache2ctl -D FOREGROUND
 ---> Running in c9d794eb983d
 ---> d941f306b9ad
Removing intermediate container c9d794eb983d
Successfully built d941f306b9ad
9d721cdbb5e49e30b253994794d4d4129883c421743dc2021506087f80562c60
```

On vérifie au travers de ces deux commandes que notre container est vulnérable à shellshock:

```
docker exec -it hitme /bin/bash -c "x='() { :;}; echo vulnerable' bash -c
"echo ceci est un test""
```

```
vulnerable
```

```
wget -qO- -U "() { test;};echo \"Content-type: text/plain\"; echo; echo;
/bin/cat /etc/passwd" http://localhost/cgi-bin/simple-cgi-bin.sh
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
```

```
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
```

On va utiliser apparmor afin d'empêcher l'exécution des scripts CGI. L'installation d'Apparmor sous debian 8 Jessie se fait de la manière suivante:

```
sudo apt-get install apparmor apparmor-profiles apparmor-utils
sudo perl -pi -e 's,GRUB_CMDLINE_LINUX="(.*)"$,\
GRUB_CMDLINE_LINUX="$1 apparmor=1 security=apparmor",' /etc/default/grub
# Pas de démarrage si la policy du binaire docker est activée
ln -s /etc/apparmor.d/usr.bin.docker /etc/apparmor.d/disable/usr.bin.docker
```

On modifie le fichier /etc/apparmor.d/docker-default et on y rajoute la ligne

deny /usr/lib/cgi-bin/** rwklx,

On recharge la politique Apparmor modifiée

`bash apparmor_parser -R /etc/apparmor.d/docker apparmor_parser -a /etc/apparmor.d/docker`

On recréé un container en utilisant la règle apparmor nouvellement créée et on vérifie que rien ne s'affiche:

```
 docker run -d  --security-opt apparmor=docker-default --publish=81:80  --
name toolate --hostname toolate jmp/shellshockme

wget -qO- -U "() { test;};echo \"Content-type: text/plain\"; echo; echo;
/bin/cat /etc/passwd" http://localhost:81/cgi-bin/simple-cgi-bin.sh
```

Les utilisateurs du groupe Docker doivent être considérés comme des utilisateurs privilégiés

Docker dispose d'une fonctionnalité (utile) : les volumes. Ces derniers lui permettent de rendre visibles des directories de l'hôte au container. L'utilisateur du groupe docker peut ainsi accéder à toute l'arborescence de l'hôte. On va ici permettre au container de copier le fichier sh et de s'adjuger les droits root.

```
# recopie de sh dans /fullaccess du container qui est aussi ./fullaccess de
l'hôte
docker run -it --name fullaccess -v $(pwd)/fullaccess:/fullaccess debian
/bin/sh -c \
        cp /bin/sh /fullaccess/ && chmod a+s /fullaccess/sh;exit'
        ls -ltr ./fullaccess
```

Utilisation des user namespaces.

Docker présente une fonctionnalité, depuis la version 1.10 très intéressante et très attendue en terme de sécurité : la possibilité d'utiliser des "user namespaces". Si l'attaquant prend le contrôle du container Docker et que le microservice du container est porté par root, l'attaquant a accès au root de l'hôte et donc aux autres containers. Les "user namespaces" permettent d'avoir un compte dans le container qui a les droits de root dans le container et qui a les droits d'un utilisateur non privilégié sur l'hôte. Après avoir activé l'option --userns-remap=default avec le daemon docker (modifiez /lib/systemd/system/docker.service et ExecStart=/usr/bin/docker daemon -H fd:// --userns-remap=default), on va mapper les uid/gid des users dans les containers à partir des fichiers /etc/subuid et /etc/subgid.

Dans subuid on voit que dockremap donne un uid de départ 1345184 et une plage de 65536 uid possibles. Le compte root aura comme uid 0 dans le container et 1345184 sur la machine hôte.

```
# Visualiser les user namespaces:
sudo systemctl stop docker
sudo rm /var/run/docker.pid
# Lancement du daemon docker avec l'option remap
sudo  docker daemon --userns-remap=default &
# Visualisation des sous users pour le user
# dockremap
cat /etc/subuid|grep dockremap
# Lancement d'un container et visu de l'appartenance du
# daemon sshd dans le container et dans l'hôte
docker run -it --name remapssh jmp/sshd ps -ef
ps -ef|grep ssh
id root
```

Le même process sshd est vu comme appartenant à root dans le container et à 1476256 dockeremap sous l'hôte. L'utilisation de cette fonctionnalité nécessite de donner les droits à l'utilisateur du container pour accéder à des volumes sur l'hôte. L'activation de cette fonctionnalité de UsernameSpace se fait pour l'ensemble des containers au niveau du daemon Docker.

Conclusions:

Docker a été critiqué sur sa sécurité. La société a fait des efforts notables pour améliorer cette situation. Quelques CVE ont été publiées sur Docker.

Un container est-il moins sécurisé qu'une machine virtuelle ?

A mon humble avis la différence n'est pas si marquée. Si vous implémentez plusieurs serveurs web sur une machine virtuelle , il vaut mieux avoir autant de containers correspondants. Compartimenter est toujours mieux que de ne rien faire. (Même si dans le cas du Titanic, ça n'a pas servi à grand choses).

Un kernel vulnérable reste un kernel vulnérable avec ou sans containers. Il est vrai que si un seul service fonctionne sur la VM , l'isolation sera meilleure. Votre choix dépendra de la criticité de votre micro-services.

Que peux-t-on retenir ?

- Le/les process d'un container ne doivent pas appartenir au compte root si possible surtout sans l'utilisation des user namespaces.
- Si vos utilisateurs font partie du groupe docker, ils peuvent accéder à l'arborescence de l'hôte et ont les mêmes pouvoirs que le compte root. Sur un serveur il faut donc éviter de mettre des utilisateurs qui n'ont rien à faire dans le groupe Docker. Sur un poste client étudiant je n'ai pas de solutions si ce n'est d'activer/désactiver par script le daemon Docker. Mais il faut relativiser : une machine virtuelle pose le même problème car l'étudiant sera root et pourra installer et exécuter les logiciels de son choix.
- Les user namespaces permettent de limiter les droits d'accès des processus du container. Par expérience, dès qu'il y a un volume mappant une directorie sur l'hôte, la gestion des droits alourdit la gestion des containers. Il n'est pas possible d'activer cette fonctionnalité pour un container et pas un autre sur le même hôte.
- Il faut limiter la consommation de ressources par les containers sous peine de voir votre container et votre hôte ployer sous un déni de services ou une fork bomb.
- Des mécanismes complémentaires comme Apparmor/Selinux, Seccomp sont complémentaires des user namespaces.
- Les images docker chargées depuis le "Docker Hub" doivent être vérifiées et lancées sur une machine virtuelle isolée ou mieux reconstruites à partir d'images de bases vérifiées. Docker met à disposition de ses clients un scanner de vulnérabilités d'images.
- L'utilisation de serveurs bare-metal est intéressante pour limiter la surface d'attaque de l'hôte mais ce n'est pas une mesure spécifique aux containers.