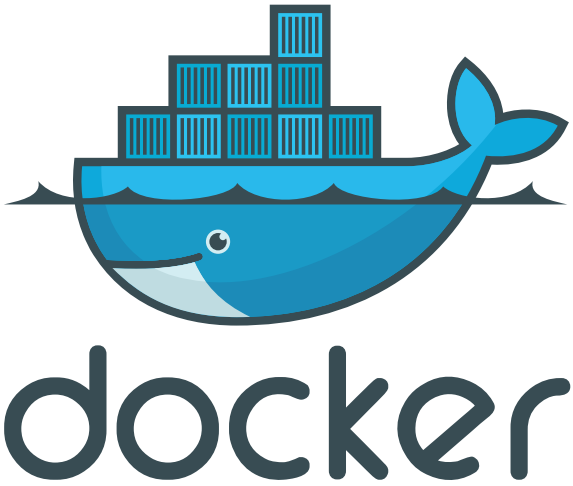

MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Projet de semestre Docker and embedded systems

Auteur :
Gary Marigliano

Encadrant :
Jean-Roland Schüler

Contact :
gary.marigliano@master.hes-so.ch

Mandant :
Haute École d'ingénierie et d'architecture de Fribourg

Version 0.0.1
7 mai 2016

Historique

Version	Date	Auteur(s)	Modifications
0.0.1	02.05.16	Gary Marigliano	Création du document

1. Résumé du document

TODO

Table des matières

2. Introduction

2.1 Contexte

Ce document est le rapport de fin de projet de semestre Docker and embedded systems. Un des buts de ce projet est de cross compiler Docker à partir de ses sources pour produire un binaire exécutable sur un ODROID-XU3 Lite (ARMv7). De plus, une partie concernant la sécurité de Docker est également traitée.

Lien : `https://github.com/krypty/docker_and_embedded_systems`

Ce projet de semestre s'inscrit dans une certaine continuité avec les projets de semestre et de bachelor de M. Loic Bassang [**?**]. Plusieurs pistes intéressantes avaient en effet été mentionnées dans ces projets là notamment une partie concernant la sécurité et Docker. Ainsi, ce rapport fera parfois des parallèles avec ces documents.

Il est important de noter que la vitesse de développement de Docker est assez hallucinante. En effet, sur Github (`https://github.com/docker/docker`) les commits se succèdent à vitesse grand V. Entre chaque version de Docker, qui sortent environ tous les mois, il est courant d'avoir plus de 3000 commits qui ont été *pushés*. Tout ceci pour dire qu'à la lecture de ce document, il est quasiment sûr que certaines pistes explorées soient définitivement obsolètes ou au contraire deviennent la voie à suivre du à une mise à jour quelconque.

2.2 Objectifs

De manière plus précise, ce projet vise à maitriser les parties suivantes :

1. Construction d'un système Linux capable de faire tourner Docker et son *daemon* en utilisant Buildroot. Pour générer le dit système, on dispose d'un *repository* Gitlab hébergé à la Haute École d'ingénierie et d'architecture de Fribourg
2. Cross compilation de Docker et de son *daemon*, capable de faire tourner des containers
3. Comprendre, analyser et évaluer l'aspect sécurité de Docker dans le cadre d'une utilisation avec une carte embarquée

Les deux premiers points ont été traités dans un précédent rapport appelé "État de l'art à la mi-projet de semestre Docker and embedded systems - Ou comment ne pas cross compiler Docker sur ARM".

Ce document se concentre, dès lors, sur le dernier point ainsi que sur le déroulement global du projet.

3. Présentation de Docker

Remarque : Si le lecteur a déjà lu le rapport "État de l'art à la mi-projet de semestre Docker and embedded systems - Ou comment ne pas cross compiler Docker sur ARM", il ne lui est pas nécessaire de relire ce chapitre sachant qu'il s'agit du même contenu.

3.1 Introduction

Le but de ce chapitre est d'apporter un contexte au projet réalisé afin de comprendre ce qui est expliqué dans la suite de ce document. Si le lecteur souhaite connaître les tréfonds de Docker, il est recommandé de lire le rapport "Docker and Internet of Things (travail de semestre)" de M. Loic Bassang [**?**].

Docker est un outil qui permet de empaqueter une application et ses dépendances dans un container léger, auto-suffisant, isolé et portable. En intégrant leur application dans un container, les développeurs s'assurent que celle-ci va tourner sur n'importe quel environnement GNU/Linux. Ainsi, le temps passé à configurer les différents environnements (développement, test et production typiquement) est réduit voire même unifié. TODO : citer `https://en.wikipedia.org/wiki/Docker_%28software%29`, `https://www.linux.com/news/docker-shipping-container-linux-code`, `https://www.docker.com/what-docker`

Les caractéristiques principales des containers Docker sont les suivantes :

- *légers* : les containers partagent le même kernel et librairies que le système d'exploitation hôte permettant ainsi un démarrage rapide des containers et une utilisation mémoire contenue
- *isolés* : les containers sont isolés grâce à des mécanismes offerts par le kernel. Voir section 3.5 pour plus de détails
- *éphémères et maintenables* : les containers sont conçus pour être créés et détruits régulièrement contrairement à un serveur ou une machine virtuelle pour lesquels un arrêt est souvent critique. Ils sont maintenables dans le sens où il est possible de revenir à une version précédente facilement et qu'il est aisé de déployer une nouvelle version

Les sections qui suivent abordent un peu plus en profondeur certains points clés de Docker qui ont été jugées importantes.

3.2 Containers vs machines virtuelles

Suite à la lecture de la section précédente, il serait normal de se dire que ces containers partagent certaines caractéristiques avec les machines virtuelles.

Voici les différences majeures qu'il existe entre les containers et les machines virtuelles[7].

Les machines virtuelles possèdent leur propre OS qui embarque ses propres binaires et librairies. Ceci engendre une perte d'espace disque importante surtout si les binaires ou libraires sont communes à plusieurs machines virtuelles. De plus, démarrer une machine virtuelle prend du temps (jusqu'à plusieurs minutes). En outre, les machines virtuelles doivent installer leurs propres drivers afin de communiquer avec l'hyperviseur (logiciel s'exécutant à l'intérieur d'un OS hôte qui gère les machines virtuelles). Un avantage cependant est l'isolation complète d'une machine virtuelle qui ne peut communiquer avec les autres par défaut.

Les containers s'exécutent de manière isolée par dessus l'OS hôte qui partage ses ressources (kernel, binaires, librairies, périphériques,...). Plus légers, les containers démarrent en quelques secondes seulement. Sur une machine, il est tout à fait possible de lancer de milliers de containers similaires car l'empreinte

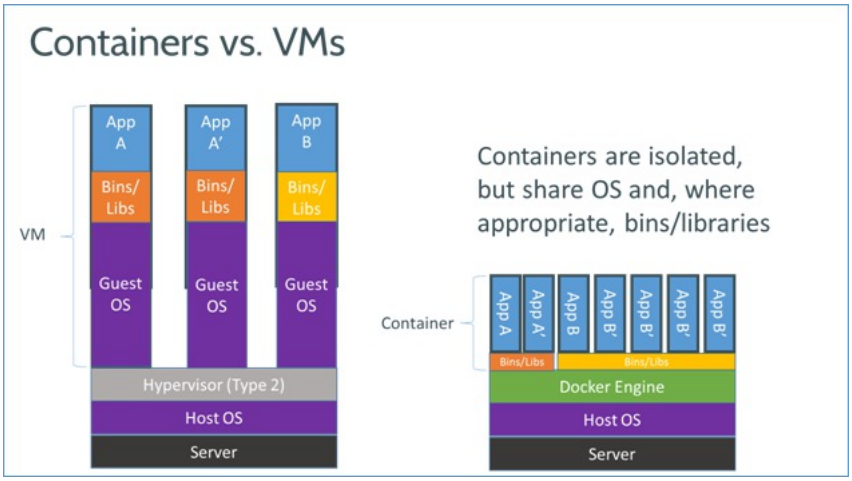

FIGURE 3.1 – Containers vs machines virtuelles

mémoire est réduite et l'espace disque occupé est partagé si les containers sont semblables. Ceci est expliqué plus en détail à la section 3.4. Les containers sont isolés mais ils peuvent aussi communiquer entre eux si on leur a explicitement donner l'autorisation.

Si le lecteur désire connaître plus de détails concernant la virtualisation, il est conseillé de lire le chapitre 3.2 du rapport "Docker and Internet of Things (travail de semestre)" de M. Loic Bassang [**?**] qui amène une bonne introduction aux différents types de virtualisation.

3.3 Docker images et Docker containers

Avec Docker, une application est encapsulée avec toutes ses dépendances et sa configuration dans une **image**.

Pour construire cette image, on utilise un Dockerfile. Il s'agit d'un fichier qui décrit les étapes de création et de configuration qu'il faut faire pour obtenir l'application configurée. C'est dans ce fichier qu'on retrouve l'OS à utiliser, les dépendances à installer et toutes autres configurations utiles au bon fonctionnement de l'application à déployer.

Typiquement un Dockerfile permettant de lancer un serveur web Nginx qui affiche un "hello world" ressemble à ceci :

```
1  FROM alpine  # image de départ
2  MAINTAINER support@tutum.co  # mainteneur du Dockerfile
3  RUN apk --update add nginx php-fpm && \  # installation des dépendances
4      mkdir -p /var/log/nginx && \
5      touch /var/log/nginx/access.log && \
6      mkdir -p /tmp/nginx && \
7      echo "clear_env = no" >> /etc/php/php-fpm.conf
8  ADD www /www  # ajout des sources de l'application
9  ADD nginx.conf /etc/nginx/  # ajout d'un fichier de configuration
10 EXPOSE 80  # ouverture du port 80
11 CMD php-fpm -d variables_order="EGPCS" && (tail -F
   ↪ /var/log/nginx/access.log &) && exec nginx -g "daemon off;" # commande
   ↪ à lancer au lancement du container
```

Source : `https://github.com/tutumcloud/hello-world/blob/master/Dockerfile`

Un Dockerfile est en quelque sorte la recette de cuisine qui permet de construire une image Docker.

Une fois l'image construite, on peut exécuter l'application dans un container. Un container Docker est donc une instance de l'image fraîchement créée. La figure 3.2 montre les relations entre un Dockerfile, une image et un container.

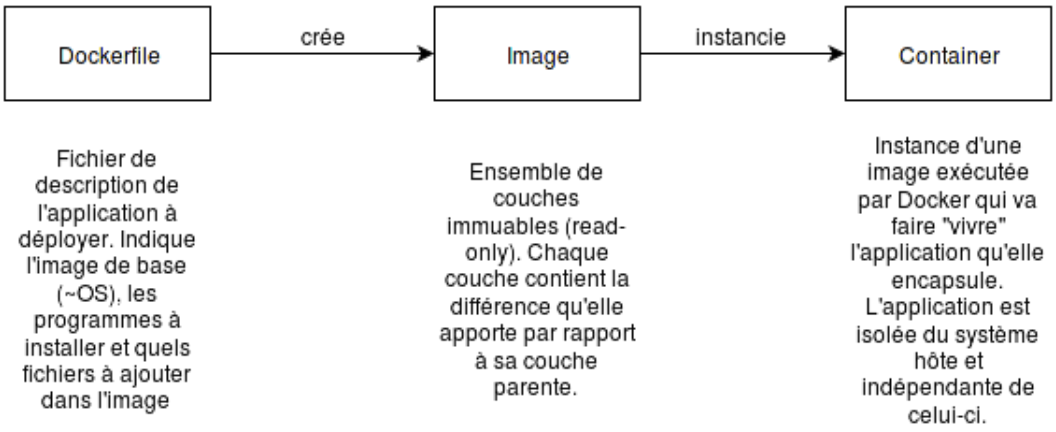

FIGURE 3.2 – Dockerfile, image et container

3.4 Système de fichiers en couche

Chaque image Docker est composée d'une liste de couches (*layers*) superposées en lecture seule[6]. Chaque couche représente la différence du système de fichiers par rapport à la couche précédente. Sur la figure 3.3, on peut voir 4 couches (identifiables avec des ID) et leur taille respectives.

FIGURE 3.3 – Couches d'une image Docker

A la création d'un container, une nouvelle couche fine est ajoutée. Cette couche, appelée "container layer" est accessible en écriture durant l'exécution du container. La figure 3.4 le montre clairement.

Un mot supplémentaire sur une nouvelle caractéristiques arrivée avec Docker 1.10 (mars 2016) ; avant cette version, Docker attribuait des UUID [1] générés aléatoirement pour identifier les couches d'une image. Désormais, ces UUID sont remplacés par des hash appelés *secure content hash*.

Les différences principales entre un UUID et un hash sont :

— Un UUID est généré aléatoirement, donc deux images exactement identiques auront un UUID différent alors qu'en utilisant un hash, le résultat sera identique

1. UUID : `https://fr.wikipedia.org/wiki/Universal_Unique_Identifier`

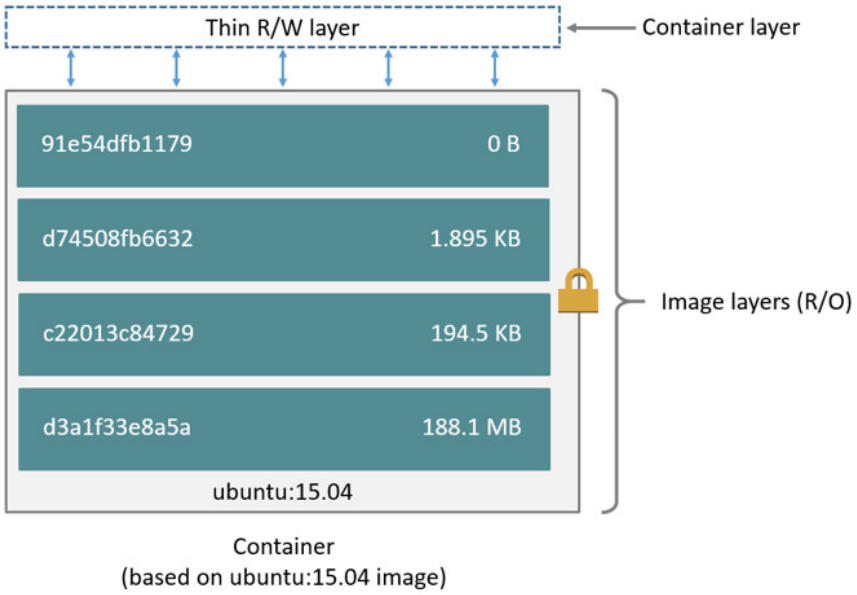

FIGURE 3.4 – Couches d'un container Docker

— Les UUID, même si la probabilité est rare [2], il est possible de générer deux fois le même UUID ce qui peut poser des problèmes lors de la construction des images
— Une image téléchargée chez une personne A aura un UUID différent que la même image téléchargée chez une personne B. Impossible de s'assurer de l'intégrité de l'image téléchargée en se basant sur l'UUID. En utilisant le hash, on s'assure du même résultat si l'image est identique

Par conséquent Docker avance les avantages suivants :

— Intégrité des images téléchargées et envoyées (sur Docker Hub par exemple)
— Évite les collisions lors de l'identification des images et des couches
— Permet de partager des couches identiques qui proviendraient de *build* différents

Le dernier point est relativement intéressant. En effet, si deux images de base (Ubuntu et Debian) sont différentes mais qu'une couche supérieure est identique (par exemple l'ajout d'un même fichier texte) alors cette couche supérieure peut être partagée entre les deux images (puisqu'elle possède le même hash). Ceci peut potentiellement offrir un gain d'espace disque conséquent si plusieurs images partagent plusieurs couches identiques. Ceci n'aurait pas pu être possible avec les UUID car les deux couches auraient produit deux UUID différents. Un exemple est visible à la figure 3.5

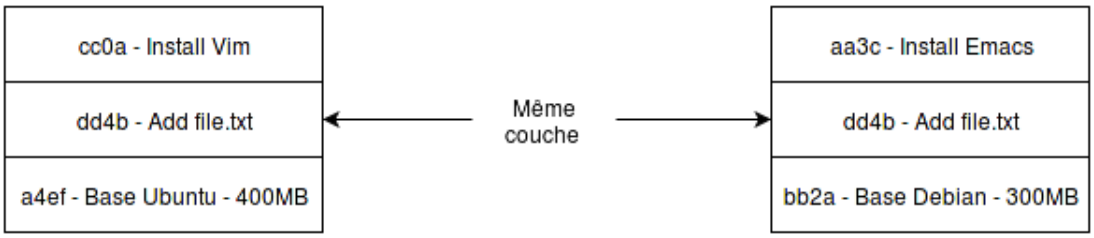

FIGURE 3.5 – Couche partagée entre deux images Docker

D'autres explications plus détaillées sur le système de fichiers en couches sont disponibles au chapitre 5.4.1 ont fait l'objet d'un chapitre dans le rapport "Docker and Internet of Things (travail de semestre)" de M. Loic Bassang.

2. Probabilité de doublon : `https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates`

3.5 Isolation

Docker met en avant le fait que ses containers soient isolés du système hôte. Pour ce faire, Docker utilise des mécanismes fournis par le kernel. On peut citer parmi ces mécanismes les *namespaces* et *cgroups*.

3.5.1 Les namespaces

Les namespaces permettent d'isoler certaines fonctionnalités d'un système d'exploitation utilisant Linux. Comme chroot permet aux processus de voir comme racine un dossier isolé du système et non pas la "vraie" racine, les namespaces isolent certains aspects du système comme les processus, les interfaces réseaux, les points de montage, etc.

Jusqu'à très récemment (docker $<$ 1.10.0), Docker supportait les namespaces suivants[2] :

- PID namespace, chaque conteneur a ses propres id de processus
- UTS namespace, pour avoir son propre hostname
- IPC namespace, qui permet d'isoler les Communications Inter-Processus
- Network namespace, chaque conteneur peux avoir sa propre interface réseau, son ip, ses règles de filtrage

Docker a désormais ajouté le support d'un nouveau namespace : user namespace. Celui-ci permet à un processus d'avoir les droits root au sein d'un namespace mais pas en dehors. Avant, Docker lançait les containers en root ce qui pouvait poser des problèmes de sécurité si un processus dans le container venait à en sortir ; il se retrouverait root sur le système hôte. Avec la prise en charge de ce namespace, un container Docker a l'impression d'être root alors qu'il n'est, en réalité, qu'un utilisateur normal sur le système hôte.

3.5.2 cgroups - Control Groups

Cgroups (control groups) est une fonctionnalité du kernel pour limiter et isoler l'utilisation des ressources (CPU, RAM, utilisation disque, utilisation réseau,...). Pour limiter les ressources, cgroups propose de créer un groupe (profil) qui décrit les limitations à respecter. Par exemple, Si on crée un groupe appelé "groupe 1" et qu'on exige de lui qu'il n'utilise au maximum 25% de la charge CPU et n'utilise au maximum 100 MB de RAM. Alors, il devient possible de lancer des programmes qui appartiennent à ce groupe et qui respectent les limites fixées.

Lorsqu'on utilise la commande `docker run` de Docker pour lancer un container, Docker peut utiliser cgroups et ainsi limiter les ressources du container [3].

3.6 Contraintes liées au monde de l'embarqué

TODO décrire les problématiques du monde embarqué, dire comment on peut en résoudre avec Docker et les pré-requis nécessaires pour utiliser Docker sur un système embarqué.

3. Docker - Runtime constraints on resources : `https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources`

4. Matériel utilisé et mise en place de la cible

Ce chapitre présente le matériel utilisé dans le projet ainsi que son installation et sa configuration de base.

Afin de réaliser ce projet, une carte embarquée ODROID-XU3 Lite a été mise à disposition afin d'y faire tourner Docker.

4.1 La carte ODROID-XU3 Lite

Cette carte possède les caractéristiques suivantes [**?**] :

- Samsung Exynos5422 Cortex™-A15 1.8Ghz quad core and Cortex™-A7 quad core CPUs
- Mali-T628 MP6(OpenGL ES 3.0/2.0/1.1 and OpenCL 1.1 Full profile)
- 2Gbyte LPDDR3 RAM at 933MHz (14.9GB/s memory bandwidth) PoP stacked
- eMMC5.0 HS400 Flash Storage
- USB 3.0 Host x 1, USB 3.0 OTG x 1, USB 2.0 Host x 4
- HDMI 1.4a for display

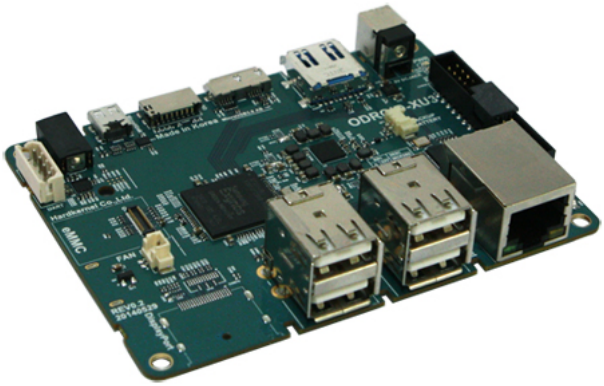

FIGURE 4.1 – ODROID-XU3 Lite

Lien vers le constructeur : `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141351880955`.

4.2 Installation du système

Initialement, il était prévu de générer un système d'exploitation minimal qui aurait été capable de faire tourner Docker et des containers. Malheureusement, il n'a pas été possible de cross compiler Docker *et son daemon* afin de lancer des containers sur ce système minimal. Plus d'informations sont disponibles dans le rapport État de l'art à la mi-projet de semestre Docker and embedded systems - Ou comment ne pas cross compiler Docker sur ARM.

Ainsi, il a été décidé, de la même manière que pour le travail de bachelor précédent, d'utiliser une distribution GNU/Linux proposant Docker dans ses packages. Le choix s'est donc porté sur **Archlinux ARM** [**?**].

Sur la page wiki de la distribution (`https://archlinuxarm.org/platforms/armv7/samsung/odroid-xu3`), on peut suivre un guide de génération de la carte SD qui contient le système d'exploitation. Une copie de ce guide est disponible à l'appendice A.

4.3 Installation de Docker

Une fois le système installé et démarré, il convient d'installer Docker depuis les dépôts de la distribution.

Ceci peut être accompli avec les commandes suivantes :

```
pacman -Syyu
reboot
pacman -S docker
systemctl enable docker
systemctl start docker
docker version
```

Sortie :

```
Client:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.6
 Git commit:   20f81dd
 Built:        Sat Mar 12 12:49:56 2016
 OS/Arch:      linux/arm

Server:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.6
 Git commit:   20f81dd
 Built:        Sat Mar 12 12:49:56 2016
 OS/Arch:      linux/arm
```

5. Objectif 3 - TODO

5.1 Situation actuelle

TODO : rappeler les objectifs, en particulier l'objectif courant (le 3)...

TODO : dire qu'on part d'une distribution Archlinux ARM et qu'on utilise un Odroid XU3 car pas réussi à cross compiler Docker. Tout comme cela avait été fait pour le travail de Bachelor précédent.

5.2 Structure de la suite du document

Pour ce projet, il a été décidé d'étudier la question de la sécurité avec Docker avec une approche en couches. A peu à la manière du modèle OSI [1] en réseau, chaque couche représente un ensemble de fonctionnalités qui, dans le cas de ce projet, doit faire l'objet d'une évaluation de la sécurité.

L'étude de la sécurité de Docker a donc été séparée avec les couches arbitraires suivantes :

- Compilation et installation de Docker : en particulier les options de compilation
- Configuration du système d'exploitation hôte : configuration du kernel, configuration des options de lancement de Docker, etc.
- Création et utilisation des images Docker : Bonnes pratiques et contraintes liées au monde de l'embarqué
- Utilisation des containers : options de lancement

Remarque : Chacune de ces couches fait l'objet d'un chapitre dans ce rapport excepté le premier point : Compilation et installation de Docker. En effet, celui-ci n'est pas traité car, comme annoncé précédemment, la cross compilation de Docker sur un système ARM n'a pas aboutie. L'effort est alors concentré sur les autres points.

1. Modèle OSI : `https://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI`

6. Configuration du système d'exploitation hôte

TODO Dans ce chapitre, on présente diverses bonnes pratiques et configurations dans le but de sécuriser Docker et/ou le système l'hébergeant.

Parmi ces techniques, on peut citer :

— TODO
— TODO

6.1 Passage en revue du benchmark de sécurité : CIS Docker 1.11.0 Benchmark

TODO décrire ce que c'est ce bench, énumerer les points testés et en explorer un certain nombre

6.1.1 Ne pas utiliser AUFS

TODO

6.1.2 User namespace

TODO

6.1.3 Interdiction de la communication réseau entre containers

TODO

6.2 Séparation des données Docker dans une partition chiffrée

TODO

6.3 TODO

TODO

7. Création et utilisation des images Docker

TODO : ce chapitre montre comment et pourquoi créer et utiliser des images Docker qui soient sécurisées et adaptées au monde de l'embarqué (taille principalement). Utilisation des hash cryptographiques pour l'OS utilisé et les versions des logiciels installés. Montrer un cas pratique avec une image commune basée sur alpine linux dire ce que ca apporte de travailler ainsi et pourquoi il faut le faire.

8. Utilisation des containers

TODO : parler de la philosophie des containers légers et éphémères, comment les exécuter de manière sûre et sécurisée, comment limiter le pouvoir des containers (conso ram, ddos, communication avec le monde extérieur ou entre containers, ...)

9. Déroulement du projet

TODO

9.1 Planning initial

TODO

9.2 Planning final

TODO

10. Proposition d'améliorations vis à vis du travail précédent

TODO : Pas prioritaire, a voir si on a le temps ! passer en revue et critique positivement le travail de Bachelor précédent. Dire que ce n'est pas une critique négative mais apporter un avis supplémentaire et plus récent (Docker évoluant beaucoup)

11. Conclusion

TODO...

Bibliographie

[1] Creating containers - Part 1, mai 2016. http://crosbymichael.com/creating-containers-part-1.html.

[2] Docker 1.10 et les user namespace, mai 2016. http://linuxfr.org/users/w3blogfr/journaux/docker-1-10-et-les-user-namespace.

[3] ODROID-XU3 | Arch Linux ARM, avril 2016. https://archlinuxarm.org/platforms/armv7/samsung/odroid-xu3.

[4] Separation Anxiety : A Tutorial for Isolating Your System with Linux Namespaces, mai 2016. https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[5] Center for Internet Security. CIS Docker 1.11.0 Benchmark, avril 2016. https://benchmarks.cisecurity.org/tools2/docker/cis_docker_1.11.0_benchmark_v1.0.0.pdf.

[6] Docker Inc. Understand images, containers, and storage drivers, mai 2016. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

[7] Docker Inc. What is Docker ?, mai 2016. https://www.docker.com/what-docker.

[8] Jaroslav. Docker 1.10, mai 2016. http://www.jrslv.com/docker-1-10/.

[9] Jean-Tiare Le Bigot. Docker for your users - Introducing user namespace, mai 2016. https://blog.yadutaf.fr/2016/04/14/docker-for-your-users-introducing-user-namespace/.

[10] Adrian Mouat. *Using Docker*. O'Reilly Media, 2005.

[11] Katherine Noyes. Docker : A 'Shipping Container' for Linux Code, mai 2016. https://www.linux.com/news/docker-shipping-container-linux-code.

[12] Prakhar Srivastav. Docker for beginners, avril 2016. http://prakhar.me/docker-curriculum/.

Appendices

A. Installation de Archlinux ARM sur ODROID-XU3 Lite

Remarque : Ce guide requiert l'utilisation d'un ordinateur sous GNU/Linux.

Source : `https://archlinuxarm.org/platforms/armv7/samsung/odroid-xu3`

A.1 Micro SD Card Creation

Replace sdX in the following instructions with the device name for the SD card as it appears on your computer.

1. Zero the beginning of the SD card :

```
dd if=/dev/zero of=/dev/sdX bs=1M count=8
```

2. Start fdisk to partition the SD card :

```
fdisk /dev/sdX
```

3. At the fdisk prompt, create the new partitions :
   a. Type o. This will clear out any partitions on the drive.
   b. Type p to list partitions. There should be no partitions left.
   c. Type n, then p for primary, 1 for the first partition on the drive, and enter twice to accept the default starting and ending sectors.
   d. Write the partition table and exit by typing w.
4. Create and mount the ext4 filesystem :

```
mkfs.ext4 /dev/sdX1
mkdir root
mount /dev/sdX1 root
```

5. Download and extract the root filesystem (as root, not via sudo) :

```
wget
 ↪ http://os.archlinuxarm.org/os/ArchLinuxARM-odroid-xu3-latest.tar.gz
bsdtar -xpf ArchLinuxARM-odroid-xu3-latest.tar.gz -C root
```

6. Flash the bootloader files :

```
cd root/boot
sh sd_fusing.sh /dev/sdX
cd ../..
```

7. (Optional) Set the MAC address for the onboard ethernet controller :
   a. Open the file root/boot/boot.ini with a text editor.
   b. Change the MAC address being set by the setenv macaddr command to the desired address.

c. Save and close the file.

8. Unmount the partition :
   umount root
9. Set the boot switches on the ODROID-XU3 board to boot from SD :
   a. With the board oriented so you can read the ODROID-XU3 on the silkscreen, locate the two tiny switches to the left of the ethernet jack.
   b. The first switch (left) should be in the off position, which is down.
   c. The second switch (right) should be in the on position, which is up.
10. Insert the micro SD card into the XU3, connect ethernet, and apply 5V power.
11. Use the serial console (with a null-modem adapter if needed) or SSH to the IP address given to the board by your router.
    — Login as the default user alarm with the password alarm.
    — The default root password is root.

A.2 eMMC Module Creation

1. Attach the eMMC module to the micro SD adapter, and plug that into your computer.
2. Follow the above steps to install Arch Linux ARM, and boot the board with the eMMC still attached to micro SD adapter, plugged into the SD slot in the board.
3. Re-flash the bootloader to the protected boot area of the eMMC module :

```
cd /boot
./sd_fusing.sh /dev/mmcblk0
```

4. Power off the board :

```
poweroff
```

5. Remove the micro SD adapter, and detach the eMMC module.
6. Set the boot switches on the ODROID-XU3 board to boot from eMMC :
   a. With the board oriented so you can read the ODROID-XU3 on the silkscreen, locate the two tiny switches to the left of the ethernet jack.
   b. The first switch (left) should be in the on position, which is up.
   c. The second switch (right) should be in the on position, which is up.
7. Connect the eMMC module to the XU3, ensuring you hear a click when doing so, connect ethernet, and apply 5V power.
8. Use the serial console (with a null-modem adapter if needed) or SSH to the IP address given to the board by your router.
   — Login as the default user alarm with the password alarm.
   — The default root password is root.