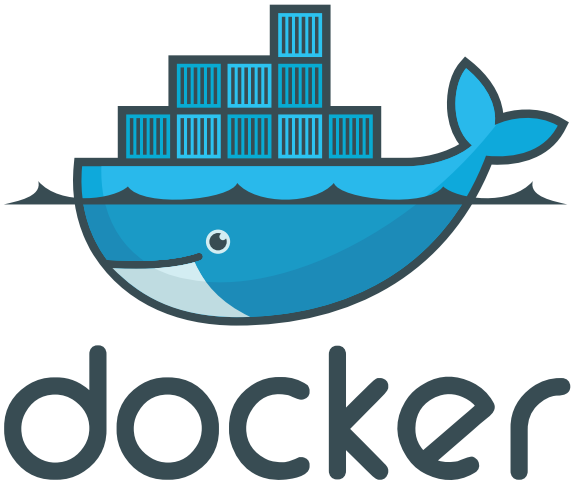


MSE | MASTER OF SCIENCE IN ENGINEERING

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 Schuler

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

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

Version 0.0.3
3 juin 2016

Historique

Version	Date	Auteur(s)	Modifications
0.0.1	02.05.16	Gary Marigliano	Création du document
0.0.2	05.05.16	Gary Marigliano	Ajout des chapitres Matériel utilisé, Présentation de Docker
0.0.3	11.05.16	Gary Marigliano	Rédaction des sections Situation actuelle, Séparation des données Docker dans une partition chiffrée, Planning

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 [28]. 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" qu'il est recommandé de lire.

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 [27].

Docker est un outil qui permet d'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[31][6][22].

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 jugés importants.

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[22].

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

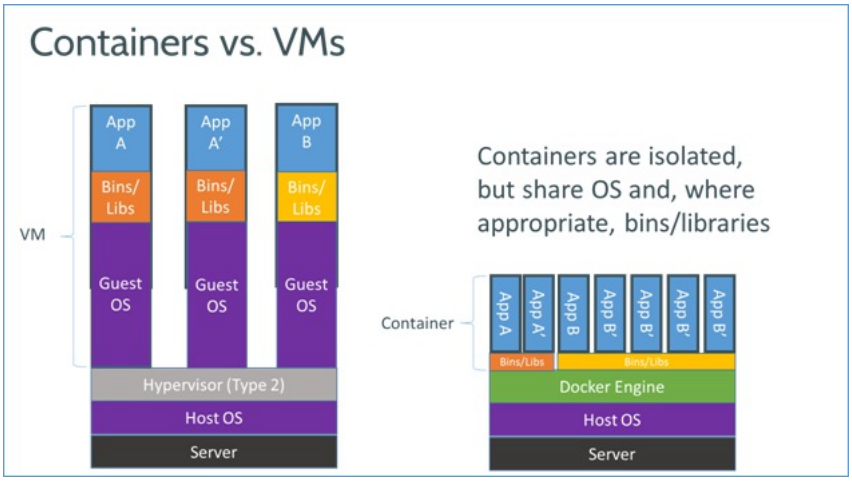


FIGURE 3.1 – Containers vs machines virtuelles

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 [27] 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 nécessaires à l'obtention de 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.

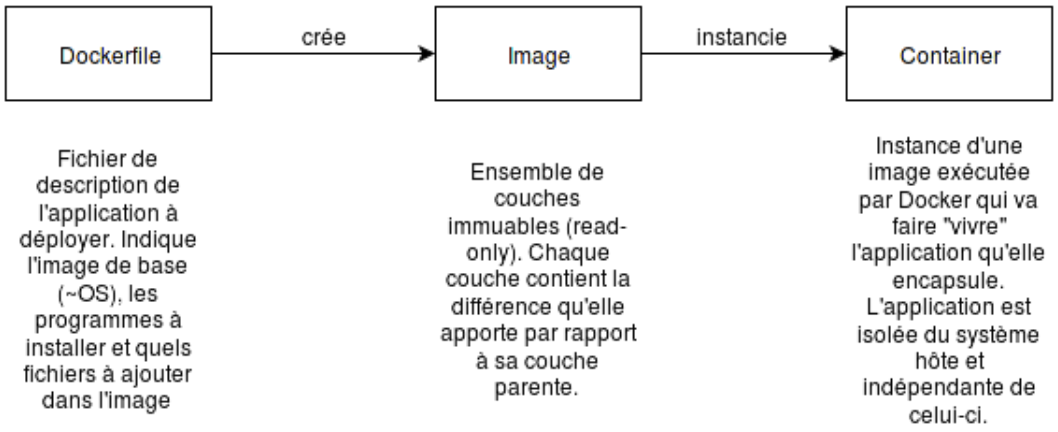


FIGURE 3.2 – Dockerfile, image et container

3.4 Système de fichiers en couche

Chaque image Docker est composée d'une liste de couches (*layers*) superposées en lecture seule[21]. 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 respective.

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 caractéristique 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
- Avec 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

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

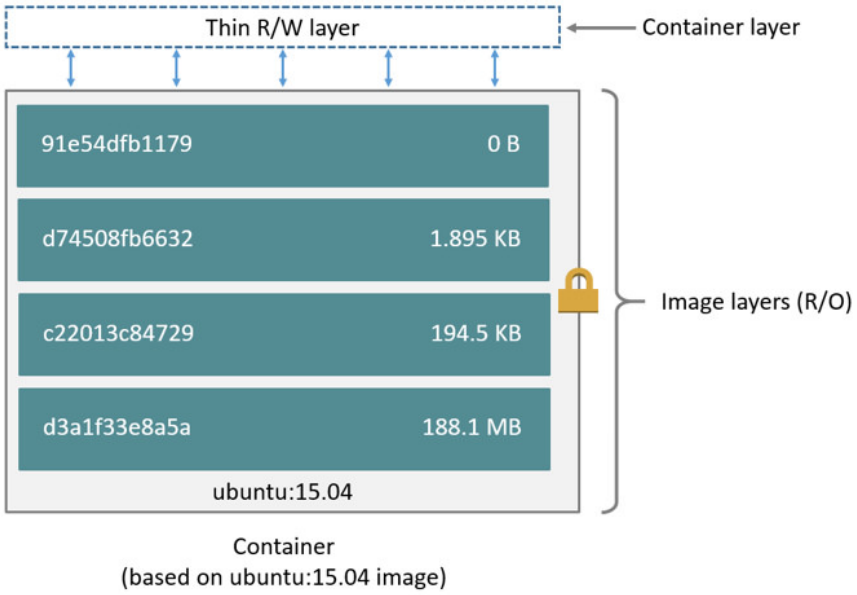


FIGURE 3.4 – Couches d'un container Docker

— 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

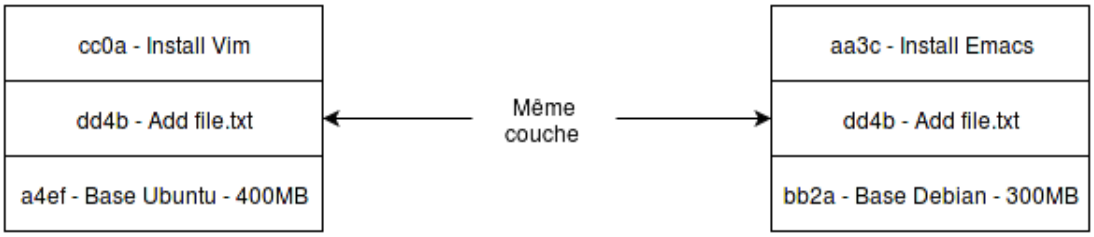


FIGURE 3.5 – Couche partagée entre deux images Docker

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

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[5] :

- 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, prioriser, isoler et contrôler 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 qu'au maximum 25% de la charge CPU et n'utilise qu'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

On entend par système embarqué, un système qui est/peut être léger, autonome, à puissance limitée, à stockage réduit, avec un OS minimal et souvent connecté.

Dans le monde de l'embarqué, il existe plusieurs problèmes récurrents lorsqu'on développe, déploie et maintient une application sur une cible. On peut citer les problèmes suivants :

- Cross-compilation souvent obligatoire
- Aucune interface graphique
- Installation et configuration des dépendances sur la cible
- Mises à jour de l'application et de ses dépendances
- Tests et journalisation (logs)
- Limiter l'utilisation en CPU, RAM et disque

Bien que ces problèmes peuvent être aussi présents dans le cas d'un développement desktop, ils ne sont pas aussi préoccupants.

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

Docker peut être utile dans le cadre d'une application embarquée car il permet une maintenance plus aisée de l'application car on peut **versionner son installation et sa configuration ainsi que celle de ses dépendances**. Ceci permet de plus facilement mettre à jour une application mais également de pouvoir revenir à une version précédente. De plus, si un accès réseau est disponible, il est même possible d'administrer Docker à distance depuis un poste de développement en se connectant au *deamon* Docker de la cible.

Cependant, il reste quelques freins et pré-requis pour pouvoir utiliser Docker sur une carte embarquée :

GNU/Linux : Il faut un système GNU/Linux et si possible une distribution qui intègre Docker dans ses packages

Images compatibles : Les images doivent être compatibles avec la plateforme de la cible. Les images x64 ne fonctionnent pas sur ARM. Actuellement, la majorité des images sont basées sur une image de base x64. Dans la plupart des cas, il suffit de trouver l'équivalent ARM de l'image de base. Par exemple, à la lecture d'un Dockerfile, il suffit de remplacer `FROM ubuntu` par `FROM armhf/ubuntu` pour que l'image arrive à se construire. Dans les autres cas, il faudra adapter les instructions du Dockerfile.

Espace disque limité : Il faut veiller à l'utilisation de l'espace disque. En effet, la plupart des images se basent sur des Ubuntu (400 MB) ou des Debian (300 MB) ce qui peut être trop volumineux pour un système embarqué. De plus plusieurs versions de ces images peuvent être téléchargées si les Dockerfiles le spécifient. Par exemple, `FROM ubuntu:14.04` pour l'image 1 et `FROM ubuntu:15.10` pour l'image 2. On favorisera l'utilisation d'une image de base légère, comme Alpine Linux [4], commune à plusieurs applications ou processus tournant sur la cible.

4. Alpine Linux : `http://www.alpinelinux.org/`

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 [9] :

- 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

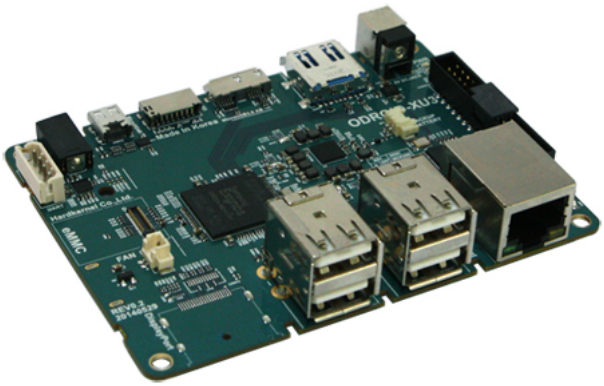


FIGURE 4.1 – ODROID-XU3 Lite

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** [1].

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.

1. Arch Linux ARM : `https://archlinuxarm.org/`

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 - Docker et la sécurité

5.1 Situation actuelle

Pour cette partie du projet, l'objectif est le suivant : Analyser et évaluer l'aspect sécurité de Docker dans le cadre d'une utilisation avec une carte embarquée.

Pour rappel, on utilise la carte ODROID-XU3 Lite . Un des objectifs de ce projet était de générer un système GNU/Linux minimal contenant un binaire Docker cross-compilé. Malheureusement, cet objectif n'a pas abouti comme expliqué 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".

Pour continuer sur la partie sécurité, il a été choisi d'utiliser une distribution GNU/Linux compatible avec la carte et qui propose dans ces dépôts un package Docker précompilé. Tout comme le travail précédent de M. Loïc Bassang, Arch Linux ARM a été sélectionné.

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

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 :

— Création d'une partition chiffrée pour stocker les données de Docker (containers et images)
— Modification des options de lancement du *daemon* Docker

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

Il s'agit d'un document à l'intention d'administrateurs systèmes ou spécialistes de la sécurités qui souhaitent intégrer, développer ou sécuriser des solutions qui utilisent Docker.

Le but de ce document est de proposer un éventail de recommandations concernant la sécurité de Docker sur plusieurs points. Voici les points évalués :

1. *Host Configuration* : Recommandations pour la configuration de la machine hôte
2. *Docker daemon configuration* : Recommandations concernant le comportement du *daemon* Docker
3. *Docker daemon configuration files* : Recommandations concernant la permission des fichiers et dossiers de configuration du *daemon* Docker
4. *Container Images and Build File* : Recommandations concernant les images de base et les fichiers de build
5. *Container runtime* : Recommandations sur les options de lancement des containers
6. *Docker Security Operations* : Rappels des bonnes pratiques concernant la sécurité de Docker

Lien : `https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.11.0_Benchmark_v1.0.0.pdf`

Dans cette partie du rapport, quelques points de ce benchmark seront analysés et appliqués à la cible dans des conditions les plus réelles possibles.

6.1.1 Situation actuelle

Avant de chercher à sécuriser la cible, il faut connaître son état de sécurisation actuel. Pour ce faire, il suffit de lancer le script du benchmark qui va exécuter toutes les recommandations proposées et indiquer quels points sont à corriger.

Avant de commencer, il faut lancer un container en arrière plan afin de pouvoir évaluer la partie *5 - Container runtime* qui évalue les containers actuellement exécutés. Pour ce faire, on choisit de lancer en tâche de fond un serveur web sur la cible.

A faire sur la cible :

```
1 docker run --name=my_container -d -p 8080:80 hypriot/rpi-busybox-httpd
```

Lancement du benchmark :

```
1 git clone https://github.com/docker/docker-bench-security.git
2 cd docker-bench-security/
3 sudo sh docker-bench-security.sh
```

On obtient la sortie suivante :

```
Initializing Sat May 14 14:12:18 UTC 2016


[INFO] 1 - Host Configuration
[WARN] 1.1  - Create a separate partition for containers
[PASS] 1.2  - Use an updated Linux Kernel
[PASS] 1.4  - Remove all non-essential services from the host - Network
[PASS] 1.5  - Keep Docker up to date
[INFO]       * Using 1.11.1 which is current as of 2016-04-27
[INFO]       * Check with your operating system vendor for support and
↪  security maintenance for docker
[INFO] 1.6  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:996:alarm
[WARN] 1.7  - Failed to inspect: auditctl command not found.
[WARN] 1.8  - Failed to inspect: auditctl command not found.
[WARN] 1.9  - Failed to inspect: auditctl command not found.
[WARN] 1.10 - Failed to inspect: auditctl command not found.
[WARN] 1.11 - Failed to inspect: auditctl command not found.
[INFO] 1.12 - Audit Docker files and directories - /etc/default/docker
[INFO]      * File not found
[INFO] 1.13 - Audit Docker files and directories - /etc/docker/daemon.json
[INFO]      * File not found
[WARN] 1.14 - Failed to inspect: auditctl command not found.
[WARN] 1.15 - Failed to inspect: auditctl command not found.


[INFO] 2 - Docker Daemon Configuration
[WARN] 2.1  - Restrict network traffic between containers
[PASS] 2.2  - Set the logging level
[PASS] 2.3  - Allow Docker to make changes to iptables
[PASS] 2.4  - Do not use insecure registries
[WARN] 2.5  - Do not use the aufs storage driver
[INFO] 2.6  - Configure TLS authentication for Docker daemon
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.7 - Set default ulimit as appropriate
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.8  - Enable user namespace support
[PASS] 2.9  - Confirm default cgroup usage
[PASS] 2.10 - Do not change base device size until needed
[WARN] 2.11 - Use authorization plugin
[WARN] 2.12 - Configure centralized and remote logging
[WARN] 2.13 - Disable operations on legacy registry (v1)


[INFO] 3 - Docker Daemon Configuration Files
[PASS] 3.1  - Verify that docker.service file ownership is set to root:root
[PASS] 3.2  - Verify that docker.service file permissions are set to 644
[PASS] 3.3  - Verify that docker.socket file ownership is set to root:root
[PASS] 3.4  - Verify that docker.socket file permissions are set to 644
[PASS] 3.5  - Verify that /etc/docker directory ownership is set to
↪  root:root
[PASS] 3.6  - Verify that /etc/docker directory permissions are set to 755
[INFO] 3.7  - Verify that registry certificate file ownership is set to
↪  root:root
[INFO]      * Directory not found
[INFO] 3.8  - Verify that registry certificate file permissions are set to
↪  444
[INFO]      * Directory not found
```

```
55  [INFO] 3.9  - Verify that TLS CA certificate file ownership is set to
    ↪  root:root
56  [INFO]      * No TLS CA certificate found
57  [INFO] 3.10 - Verify that TLS CA certificate file permissions are set to
    ↪  444
58  [INFO]      * No TLS CA certificate found
59  [INFO] 3.11 - Verify that Docker server certificate file ownership is set
    ↪  to root:root
60  [INFO]      * No TLS Server certificate found
61  [INFO] 3.12 - Verify that Docker server certificate file permissions are
    ↪  set to 444
62  [INFO]      * No TLS Server certificate found
63  [INFO] 3.13 - Verify that Docker server key file ownership is set to
    ↪  root:root
64  [INFO]      * No TLS Key found
65  [INFO] 3.14 - Verify that Docker server key file permissions are set to 400
66  [INFO]      * No TLS Key found
67  [PASS] 3.15 - Verify that Docker socket file ownership is set to
    ↪  root:docker
68  [PASS] 3.16 - Verify that Docker socket file permissions are set to 660
69  [INFO] 3.17 - Verify that daemon.json file ownership is set to root:root
70  [INFO]      * File not found
71  [INFO] 3.18 - Verify that daemon.json file permissions are set to 644
72  [INFO]      * File not found
73  [INFO] 3.19 - Verify that /etc/default/docker file ownership is set to
    ↪  root:root
74  [INFO]      * File not found
75  [INFO] 3.20 - Verify that /etc/default/docker file permissions are set to
    ↪  644
76  [INFO]      * File not found
77
78
79  [INFO] 4 - Container Images and Build Files
80  [WARN] 4.1  - Create a user for the container
81  [WARN]      * Running as root: my_container
82  [WARN] 4.5  - Enable Content trust for Docker
83
84
85  [INFO] 5  - Container Runtime
86  [WARN] 5.1  - Verify AppArmor Profile, if applicable
87  [WARN]      * No AppArmorProfile Found: my_container
88  [WARN] 5.2  - Verify SELinux security options, if applicable
89  [WARN]      * No SecurityOptions Found: my_container
90  [PASS] 5.3  - Restrict Linux Kernel Capabilities within containers
91  [PASS] 5.4  - Do not use privileged containers
92  [PASS] 5.5  - Do not mount sensitive host system directories on containers
93  [PASS] 5.6  - Do not run ssh within containers
94  [PASS] 5.7  - Do not map privileged ports within containers
95  [PASS] 5.9 - Do not share the host's network namespace
96  [WARN] 5.10 - Limit memory usage for container
97  [WARN]      * Container running without memory restrictions: my_container
98  [WARN] 5.11 - Set container CPU priority appropriately
99  [WARN]      * Container running without CPU restrictions: my_container
100 [WARN] 5.12 - Mount container's root filesystem as read only
101 [WARN]      * Container running with root FS mounted R/W: my_container
102 [WARN] 5.13 - Bind incoming container traffic to a specific host interface
103 [WARN]      * Port being bound to wildcard IP: 0.0.0.0 in my_container
104 [WARN] 5.14 - Set the 'on-failure' container restart policy to 5
105 [WARN]      * MaximumRetryCount is not set to 5: my_container
106 [PASS] 5.15 - Do not share the host's process namespace
107 [PASS] 5.16 - Do not share the host's IPC namespace
```

```
[PASS] 5.17 - Do not directly expose host devices to containers
[INFO] 5.18 - Override default ulimit at runtime only if needed
[INFO]      * Container no default ulimit override: my_container
[PASS] 5.19 - Do not set mount propagation mode to shared
[PASS] 5.20 - Do not share the host's UTS namespace
[PASS] 5.21 - Do not disable default seccomp profile
[PASS] 5.24 - Confirm cgroup usage
[WARN] 5.25 - Restrict container from acquiring additional privileges
[WARN]      * Privileges not restricted: my_container


[INFO] 6  - Docker Security Operations
[INFO] 6.4 - Avoid image sprawl
[INFO]      * There are currently: 2 images
[INFO] 6.5 - Avoid container sprawl
[INFO]      * There are currently a total of 8 containers, with 1 of them
↪  currently running
```

Comme on peut le voir, il y a plusieurs points qui sont en *WARN*. Pour la suite du document, plusieurs de ces points vont être analysés et mitigés.

6.1.2 1.1 - Create a separate partition for containers

On recommande pour ce point de créer une partition séparée pour les containers. Ce point est traité à la section 6.2. Dans cette section, une étape supplémentaire est proposée ; la partition est chiffrée.

6.1.3 2.1 - Restrict network traffic between containers

Par défaut, le trafic réseau est autorisé entre containers du même hôte. On devrait désactiver ce comportement par défaut et explicitement autoriser une communication entre containers grâce au "linking".

Pour remédier à ceci, il faut lancer le *daemon* Docker avec cette option : `docker daemon -icc=false`

A faire sur la cible :

```
sudo vim /usr/lib/systemd/system/docker.service
```

Ajouter l'option icc=false à l'instruction ExecStart. Le résultat devrait ressembler à ceci :

```
ExecStart=/usr/bin/docker daemon -H fd:// --icc=false
```

Redémarrage du daemon :

```
sudo systemctl daemon-reload # recharge la configuration des services
sudo systemctl restart docker # relance le service docker
```

Résultat du benchmark après cette modification : `[PASS] 2.1 - Restrict network traffic between containers`.

6.1.4 2.5 - Do not use the aufs storage driver

Pour stocker les images et les différentes couches qui les composent, Docker utilise un driver de stockage[21], [18], [26]. Un driver de stockage (*driver storage*) est l'implémentation d'une abstraction de services liés à la gestion des couches des images et de la couche inscriptible (*writable*) des containers.

L'utilisation d'un driver ou d'un autre est transparent pour l'utilisateur. Cependant, une certaine implémentation, aufs, est considérée peu stable et peut causer des crashes au niveau du kernel, voir [20] page 45. Ce driver est donc à remplacer par un autre driver de stockage comme devicemapper par exemple.

Inconvénient : aufs est le seul driver de stockage qui permet aux containers de partager la mémoire des librairies partagées. Cela peut être utile si la cible fait tourner des milliers de containers utilisant les mêmes programmes ou librairies [20].

Dans le cadre d'une carte embarquée, on ne va pas faire tourner des milliers de containers en même temps et par conséquent changer de driver storage n'est pas un problème.

Par défaut Docker utilise devicemapper mais suivant la distribution ou la plateforme utilisée, un autre storage driver est utilisé. Dans le cas présent, aufs est malheureusement utilisé.

Vérification que device-mapper est présent sur le système :

```
[alarm@alarm docker-bench-security]# pacman -Qi device-mapper
Name            : device-mapper
Version         : 2.02.149-1
Description     : Device mapper userspace library and tools
Architecture    : armv7h
URL             : http://sourceware.org/dm/
Licenses        : GPL2  LGPL2.1
Groups          : base
Provides        : None
Depends On      : glibc  systemd
Optional Deps   : None
Required By     : cryptsetup  docker  lvm2
Optional For    : None
Conflicts With  : None
Replaces        : None
Installed Size  : 871.00 KiB
Packager        : Arch Linux ARM Build System
 ↪ <builder+xu2@archlinuxarm.org>
Build Date      : Sun Apr 17 12:52:54 2016
Install Date    : Mon Apr 18 14:12:44 2016
Install Reason  : Explicitly installed
Install Script  : No
Validated By    : SHA-256 Sum
```

Comme avant, on change les paramètres de lancement du *daemon* Docker :

```
sudo vim /usr/lib/systemd/system/docker.service
```

Ajouter l'option `-storage-driver=devicemapper` à l'instruction ExecStart. Le résultat devrait ressembler à ceci :

```
ExecStart=/usr/bin/docker daemon -H fd:// --icc=false
 ↪ --storage-driver=devicemapper
```

```
sudo systemctl daemon-reload # recharge la configuration des services
sudo systemctl restart docker # relance le service docker
```

6.1.5 2.8 - Enable user namespace support

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 :
- 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

Pour plus de détails, voir section 3.5.

Il y a peu de temps, Docker ajouté le support d'un nouveau namespace[5] : 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.

Pour mettre en place ce mécanisme, il faut réaliser les opérations suivantes :

```
1 touch /etc/subuid /etc/subgid
```

Et ajouter l'option `-userns-remap=default` au démarrage du *daemon* Docker.

Malheureusement, il a été impossible de démarrer le *daemon* avec cette option que ce soit sur la cible (Archlinux, kernel 3.10.96), sur un portable Archlinux (kernel 4.5.4) ou sur un autre portable Xubuntu 16.04 (kernel 4.4.0). Le *daemon* refusait de démarrer ou se bloquait. Il semblerait que cela ne soit pas un cas isolé. En effet plusieurs issues (#20751, #21130 [1]) ont été ouvertes sur le *repository* Github de Docker.

6.1.6 2.11 - Use authorization plugin

Le modèle de permission Docker par défaut est tout ou rien. Tout utilisateur qui a accès au *daemon* Docker peut effectuer n'importe quelles opérations sur tous les containers. Dans le cas où l'on a besoin d'un contrôle plus fin, il est possible de créer des plugins d'autorisation (*authorization plugins*) et les ajouter à la configuration du *daemon* Docker. Il est, par exemple, possible de définir que l'utilisateur Bob ne puisse pas arrêter les containers mais uniquement les lister.

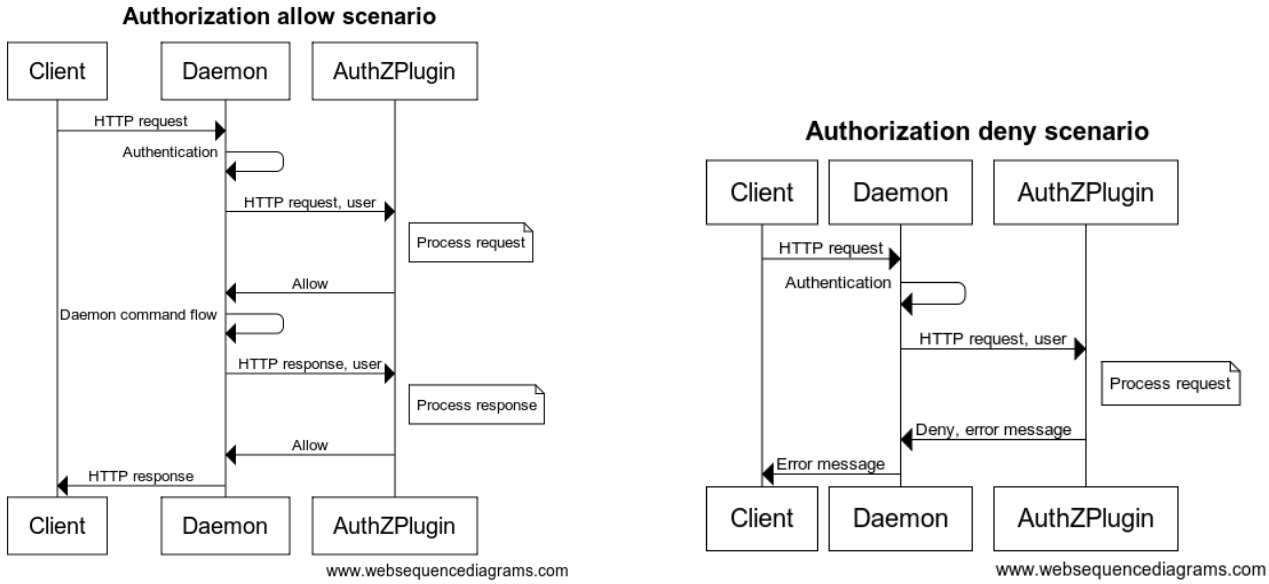
Un plugin d'autorisation approuve ou refuse les requêtes du client (un utilisateur) vers le *daemon* en se basant sur le contexte actuel d'authentification et le contexte de commande. Le contexte d'authentification contient toutes les détails de l'utilisateur et de la méthode d'authentification et le contexte de commande contient toutes les données utiles à la requête de l'utilisateur.

Remarque : La communication entre le *daemon* et le client[19] se fait à travers un socket Unix (`/var/run/docker.sock`) mais peut aussi se faire via un socket TCP (`tcp://0.0.0.0:2375`) pour accéder au *daemon* de manière distante. Grâce à ce mécanisme, Docker propose *Docker Remote API*[16] une API *REST-like* qui permet d'accéder à la liste des containers, des images, de créer des containers, etc.

1. Docker issues : `https://github.com/docker/docker/issues`

Pour créer un plugin d'autorisation, il faut respecter le contrat décrit par la *Docker Plugin API*. De manière similaire à la *Docker Remote API*, cette API définit un cadre afin d'étendre les possibilités de Docker comme par exemple autoriser ou refuser l'accès à certaines fonctionnalités de Docker pour un utilisateur ou un groupe donné.

Maintenant les notions de communication client *daemon* et API REST éclaircies, on peut enfin voir l'architecture de base des plugins d'autorisation.



(a) Cas où la requête est autorisée

(b) Cas où la requête est refusée

FIGURE 6.1 – Scénarios d'autorisation Docker [2]

Comme on peut le voir sur les figures ci-dessus, les requêtes sont des requêtes HTTP JSON comme on peut le trouver fréquemment dans avec les API REST.

Il est possible d'utiliser plusieurs plugins en même temps. Dans ce cas, pour qu'une requête soit acceptée, il faut qu'elle puisse traverser tous les plugins (donc être approuvée par tous les plugins). Si un seul plugin refuse la requête alors elle est refusée.

La création et mise en place de plugin n'est pas présenté dans ce rapport. Néanmoins, voici un exemple de plugin : `https://github.com/twistlock/authz`.

Analyse : Dans le cas d'une utilisation embarquée, il n'est pas indispensable d'avoir recours à un contrôle plus fin de l'utilisation de Docker car :

- la cible est souvent mono-utilisateur
- les containers appartiennent tous au même utilisateur. Dans le cas d'une plateforme offrant un service de type CaaS (Containers as a Service), on peut comprendre que chaque utilisateur n'ait accès qu'aux containers qu'il lance.

Par conséquent, on décide de ne pas mettre en place ce mécanisme. On rappelle que ce benchmark a surtout pour but de montrer un ensemble de bonnes pratiques et de rendre attentif à certains aspects de la sécurité avec Docker. Il n'est néanmoins pas toujours indispensable d'appliquer toutes les mesures de sécurités si on en exprime pas le besoin.

2. Source des images : `https://docs.docker.com/engine/extend/plugins_authorization/`

6.1.7 2.12 - Configure centralized and remote logging

Docker supporte plusieurs *log drivers* (manière de journaliser). Par défaut, les logs des containers sont produits dans des fichiers JSON sur la machine hôte. Mais il est recommandé de centraliser les logs sur une machine distante pour voir et traiter les logs globalement.

Le benchmark CIS recommande d'externaliser les logs en optant pour le log driver *syslog*. *Syslog*[32][11] est un protocole définissant un service de journaux d'évènements d'un système informatique. Son but est de transporter par réseau les messages de journalisation générés par une application (dans ce cas Docker) vers un serveur hébergeant un serveur *Syslog*. Le but étant de regrouper tous les journaux au même endroit.

Pour la suite, on va mettre en place un petit exemple de démonstration qui n'a absolument pas la prétention d'être exhaustif ni sécurisé. Cet exemple a uniquement pour but de montrer un cas simple de *remote logging*.

Pour cet exemple, il y a à disposition un ordinateur portable qui fait office de serveur *Syslog* et la cible Odroid qui va lancer un container dont les logs sont envoyés à l'ordinateur portable. On peut voir cette configuration ci-dessous.

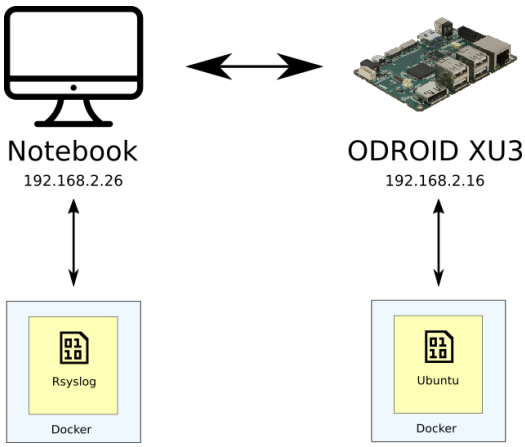


FIGURE 6.2 – Structure de démonstration Syslog

Le container (ici Ubuntu) va être lancé en indiquant que les logs doivent être envoyés au PC portable disposant d'un serveur syslog dans un container appelé rsyslog.

Pour le serveur *Syslog*, un container Docker prêt à l'emploi va être utilisé. Ce container utilise une image Rsyslog dont la source se trouve ici : `https://hub.docker.com/r/panoptix/rsyslog/`.

Lancement du container Rsyslog sur le portable et affichage des logs distants :

```
# démarrage du container
docker run --name rsyslog -t -d -p 514:514/udp -p 514:514 panoptix/rsyslog

# affichage des logs en temps réel
docker exec -it rsyslog bash
tail -f /var/log/syslog
```

Lancement du container Ubuntu (producteur de logs) sur la cible :

```
docker run -d --log-driver=syslog --log-opt
↪  syslog-address=tcp://192.168.2.26 --log-opt tag="remote" armhf/ubuntu
↪  echo "hello from Odroid"
```

Ce container va lancer la commande `echo` et se stopper immédiatement après. Évidemment dans un cas réel, le container effectuera des opérations plus longues et produirait un plus grande quantité de logs mais ici le but est de voir le principe général.

Sur l'ordinateur portable, on peut observer l'affichage suivant :

```
May 22 18:23:42 fc6dca22947a rsyslogd: [origin software="rsyslogd"
↪  swVersion="8.11.0" x-pid="1" x-info="http://www.rsyslog.com"] start
May 22 18:38:09 192.168.2.16 docker/remote[284]: hello from Odroid
```

Comme on peut le voir, on est capable de récupérer les logs de la cible sur l'ordinateur portable.

Remarque : Pour cet exemple, les options de logging ont été précisés au niveau du lancement du container (`-log-driver=XXX -log-opt syslog-address=tcp://aaa.bbb.ccc.ddd -log-opt YYY`). Idéalement, il faudrait définir par défaut toutes ces options au démarrage du *daemon*. Pour ce faire, il suffit de modifier la configuration de lancement du *daemon* comme il a été montré aux sections précédentes. Par exemple, on pourrait configurer le *daemon* ainsi :

```
ExecStart=/usr/bin/docker daemon -H fd:// --icc=false
↪ --storage-driver=devicemapper --log-driver=XXX --log-opt
↪  syslog-address=tcp://aaa.bbb.ccc.ddd --log-opt YYY
```

Analyse : On l'a vu les containers produisent des logs qu'il est recommandé de centraliser sur une machine distante. Pour ce faire, Docker propose plusieurs *log drivers* qui permettent ce besoin. Dans cet exemple, *Syslog* a été utilisé mais il existe d'autres *log drivers*[3].

L'exemple présenté est sommaire mais il montre le principe général. Dans un cas plus pratique, par exemple un ensemble de cibles qui transmettent la température ambiante à un serveur (ou autre application IoT), on configurerait les cibles pour émettre leurs logs sur un serveur distant. Ce serveur disposerait ensuite d'une interface web affichant l'ensemble des logs des cibles et il serait ainsi possible de suivre le fonctionnement de toutes les cibles. De plus, dans un cas pratique, il faudrait aussi veiller à regarder le contenu de l'image rsyslog téléchargée et vérifier qu'elle fasse bien uniquement ce qu'on lui demande et rien de plus ou alors créer une image rsyslog de toutes pièces.

6.1.8 2.13 - Disable operations on legacy registry (v1)

Un *registry* est un service responsable de l'hébergement et de la distribution d'images Docker. C'est une sorte d'App Store pour image Docker. Le *registry* par défaut est Docker Hub[4]. Il est également possible d'utiliser son propre *registry*. Dans tous les cas, il est recommandé de ne plus utiliser la version 1 de l'API de communication avec les *registries* car la version 2 permet notamment le transfert sécurisé de données en utilisant TLS[12].

Pour forcer Docker a n'utiliser que des *registries* v2, il suffit d'ajouter l'option `-disable-legacy-registry` au démarrage du *daemon* Docker.

A faire sur la cible :

```
sudo vim /usr/lib/systemd/system/docker.service
```

Ajouter l'option icc=false à l'instruction ExecStart. Le résultat devrait ressembler à ceci :

```
ExecStart=/usr/bin/docker daemon -H fd:// --icc=false
↪ --storage-driver=devicemapper --disable-legacy-registry
```

3. Docker log drivers : `https://docs.docker.com/engine/admin/logging/overview/`
4. Docker Hub : `http://hub.docker.com/`

Redémarrage du daemon :

```
sudo systemctl daemon-reload # recharge la configuration des services
sudo systemctl restart docker # relance le service docker
```

Résultat du benchmark après cette modification : `[PASS] 2.13 - Disable operations on legacy registry (v1)`.

6.1.9 4.1 - Create a user for the container

Par défaut, lorsqu'on lance un container, l'utilisateur à l'intérieur de ce container est root. Pour des raisons évidentes de sécurité, root n'est pas toujours l'utilisateur le plus adapté quand on lance un container. De plus, si par une quelconque raison, on arrive à sortir du container, on se retrouve root sur la machine hôte !

Pour mitiger ces effets, il est possible de :
- Créer un utilisateur avec des droits limités dans l'image du container (`USER toto` dans le Dockerfile)
- Spécifier l'utilisateur à utiliser lors du lancement du container (`docker run -user toto`)
- Faire croire au container qu'il est root alors qu'en réalité il s'agit d'un utilisateur à droits limités (user namespace)

La suite a pour but de montrer les différentes techniques pour changer l'utilisateur au sein du container.

État initial

On utilise l'image Alpine sans modifications et on observe les résultats suivants :

```
docker run -it alpine /bin/sh
/ # whoami
root
/ # touch /root.txt
/ # ls /root.txt
/root.txt
/ # ls -l /root.txt
-rw-r--r--    1 root     root             0 May 23 13:41 /root.txt
```

On voit que par défaut l'utilisateur dans le container est root. Cet utilisateur peut donc écrire un fichier à la racine.

Dans le Dockerfile

Pour l'exemple qui suit, on va créer une image qui utilise Alpine Linux comme base. Cette image est modifiée pour qu'elle se lance par défaut avec un utilisateur toto qui a des droits limités.

```
mkdir alpine-user
vim Dockerfile
```

Dans le Dockerfile, on écrit :

```
FROM alpine

# tâches nécessitant un accès root
# par exemple, installer des packages
RUN apk update && apk add vim
```

```
6
7   # fin des tâches demandant un accès root
8   # passage à un user à droits limités
9   RUN adduser -h /home/toto -s /bin/sh -D toto
10  USER toto
11
12  CMD ["/bin/sh"]
```

On utilise donc l'image de base d'Alpine sur laquelle on effectue des tâches administratives nécessitant root. Une fois ces tâches effectuées, il n'est plus nécessaire d'être root. On crée alors un utilisateur à droit limité appelé toto. C'est cet utilisateur qui sera utilisé par défaut au lancement d'un container utilisant cette image.

```
1  docker run -it alpine-user
2  / whoami
3  toto
4  / touch /toto.txt
5  touch: /toto.txt: Permission denied
```

Ici, on voit que l'utilisateur en cours est toto et qu'il n'a pas les droits root.

Au lancement du container

Il est également possible de spécifier l'utilisateur à utiliser au lancement du container grâce à l'option `-user`.

Les cas suivants sont testés :

- root : on peut spécifier root si l'image a défini un autre utilisateur par défaut. Utilisation déconseillée.
- nobody : l'utilisateur nobody est un utilisateur aux droits très limités et la plupart des distributions l'intègre de base donc pas besoin le créer expressément.
- gary : cas où l'utilisateur existe sur l'hôte mais pas dans le container
- toto : cas où l'utilisateur n'existe pas sur l'hôte mais dans le container

Utilisateur root :

```
1  > docker run --user root -it alpine-user
2  / # whoami
3  root
4  / # touch /root.txt
5  / # ls -al /root.txt
6  -rw-r--r--    1 root     root             0 May 23 19:35 /root.txt
```

Le container est lancé en root. On peut écrire à la racine sans souci.

Utilisateur root :

```
1  > \$ docker run --user nobody -it alpine-user
2  ~ \$ whoami
3  nobody
4  ~ \$ touch /nobody.txt
5  touch: /nobody.txt: Permission denied
```

Le container est lancé avec l'utilisateur nobody (présent sur l'OS hôte et surtout dans le container). L'écriture sur `/` est refusée.

Utilisateur gary :

```
> docker run --user gary -it alpine-user
docker: Error response from daemon: linux spec user: Unable to find user
↪ gary.
```

Lancer un container avec un utilisateur qui n'existe pas dans le container n'est pas possible.

Utilisateur toto :

```
> \$ docker run --user toto -it alpine-user
/ \$ whoami
toto
/ \$ touch /toto.txt
touch: /toto.txt: Permission denied
```

Le container est lancé avec l'utilisateur toto (absent sur l'OS hôte et mais présent dans le container). L'écriture sur `/` est refusée.

Analyse

On remarque que par défaut, les containers sont lancés en root. Si le container n'a pas besoin d'avoir des accès root, il est fortement recommandé de construire une image mettant en place un utilisateur à droits limités. Dans le cas où l'on utilise une image écrite par quelqu'un d'autre, on peut essayer de lancer avec l'utilisateur nobody si on estime qu'elle n'a pas besoin des droits root pour fonctionner. Évidemment, cela ne fonctionnera pas si l'image a réellement besoin des droits root comme par exemple pour installer des packages.

Le dernier cas concerne l'utilisation du user namespace. Ici c'est différent, on mappe un utilisateur standard de la machine hôte à l'utilisateur root du container. Ainsi, on peut effectuer des tâches nécessitant les droits root dans le container. Cependant, si le processus au sein du container vient à s'en échapper, il n'est pas root mais l'utilisateur mappé aux droits limités.

6.1.10 4.5 - Enable Content trust for Docker

Rappel : Un *registry* (Docker Hub par exemple) héberge des *repositories* et un *repository* contient plusieurs images. Chacune de ses images est une "couche" de l'application ou du processus proposé(e) par un utilisateur.

Docker Content Trust[15][30] est une fonctionnalité de Docker arrivée avec la version 1.8 qui permet de vérifier les auteurs des images Docker. Avant qu'un auteur publie une image sur un *registry* distant (par exemple Docker Hub), Docker Engine signe l'image en local avec la clé privée de l'auteur. Quand l'image est rapatriée (*pulled*), Docker Engine compare la clé publique contenue dans le certificat qui accompagne l'image avec la clé publique de l'auteur pour vérifier que l'image téléchargée n'a pas été altérée et est à jour.

Docker Content Trust n'est pas activé par défaut. Une fois activé, ces mécanismes de vérifications et de contrôles sont transparents pour l'utilisateur Docker. Les commandes `docker pull`, `docker push`, `docker build`, `docker create` et `docker run` sont utilisées de la même façon à la différence qu'elles opèrent sur du contenu signé.

Voici dans les grandes lignes comment fonctionne ce système :

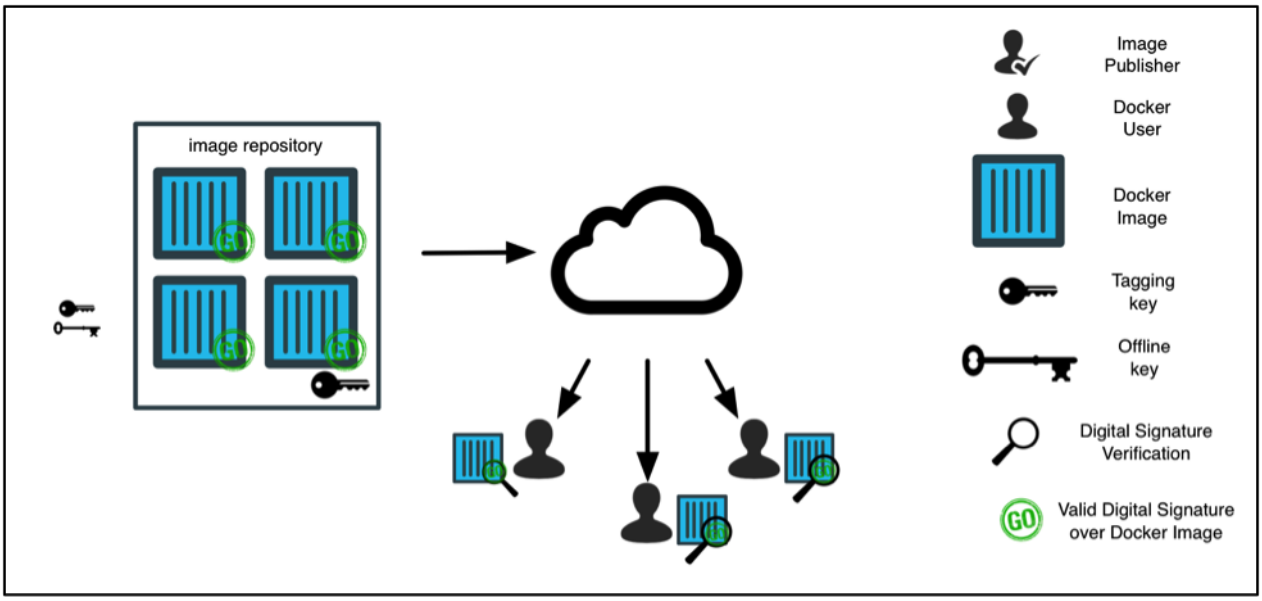


FIGURE 6.3 – Vue d'ensemble de Docker Content Trust

Soient deux utilisateurs Docker, Alice et Bob. Alice est une utilisatrice qui met à disposition des images dans un *repository* sur un *registry* (Docker Hub, par exemple). Bob est un utilisateur qui veut télécharger (comprendre *pull*) les images qu'Alice a mises à disposition.

Quand Alice désire publier une image dans un de ses *repositories* appelé AliceSoft, deux clés sont générées, la *Tagging Key* et la *Offline Key*. Ces deux clés sont détaillées plus bas. Les images sont signées en utilisant la *Tagging Key* qui est signée avec la *Offline Key*.

Bob désire ensuite télécharger cette image. Comme c'est la première fois que Bob télécharge une image du *repository* AliceSoft, Docker Engine va télécharger en HTTPS le certificat de ce *repository*. Les fois suivantes, on réutilise le certificat téléchargé. Puis, Docker Engine compare la clé contenue dans le certificat avec la clé de l'image téléchargée.

Docker Content Trust utilise deux clés distinctes, une *Offline Key* et une *Tagging Key* qui sont générées la première fois qu'un auteur pousse une image. Elles sont stockés côté client (chez Alice dans l'exemple). Chacun de ses *repositories* possède une *Tagging Key* unique. La première fois que Bob veut télécharger une image d'Alice, il établit une confiance avec le *repository* d'Alice en utilisant la *Offline Key*.

— **La Tagging Key** : est générée pour chaque nouveau *repository* qu'un auteur possède. Elles peuvent être exportées et partagées avec toute personne ou système qui exprime le besoin de signer du contenu (typiquement une image) dans ce *repository*, par exemple, un *repository* appartenant à un groupe de personnes.
— **La Offline Key** : est la clé la plus importante car c'est elle qui permet d'établir la confiance pour le *repository* ou les *repositories* qu'elle contrôle. Cette clé est aussi utilisée pour effectuer des rotations sur les autres clés existantes (si une clé a été compromise par exemple). La *Offline Key* est secrète, doit être stockée dans un endroit sûr et doit également être sauvegardée.

pourquoi on le change

Grâce à ces protections, Docker annonce une meilleure sécurité quant à la provenance des images téléchargées. On évite ainsi plusieurs types d'attaque comme :
— Man In the Middle (l'attaquant se place entre le *registry* et l'utilisateur et altère l'image téléchargée)
— Les attaques par rejeu (un attaquant force l'utilisateur à télécharger une image valide mais ancienne dont il sait qu'elle contient des vulnérabilités connues)
— Les clés compromises (l'attaquant à volé la clé ou l'a cassée)

Ces garanties sont justifiées sur le blog de Docker à l'adresse suivante : `https://blog.docker.com/2015/08/content-trust-docker-1-8/`.

Mise en place

Pour activer Docker Content Trust, il suffit de définir la variable d'environnement `export DOCKER_CONTENT_TRUST=1`. Définir une variable d'environnement dans un terminal est valable durant la durée de la session et pour l'utilisateur courant. Pour définir cette variable de manière permanente et pour tous les utilisateurs, il peut modifier le fichier `/etc/environment` et ajouter la variable d'environnement voulue.

```
sudo vim /etc/environment
```

Et ajouter :

```
#
# This file is parsed by pam_env module
#
# Syntax: simple "KEY=VAL" pairs on separate lines
#
DOCKER_CONTENT_TRUST=1
```

Après avoir redémarré, on relance le benchmark et on obtient la sortie suivante : `[PASS] 4.5 - Enable Content trust for Docker`

Utilisation dans la pratique

Désormais, tous les `docker pull` que l'on fait exige que les images téléchargées soient signées.

Il existe donc 3 cas :

1. L'image n'est pas signée et fait une erreur
2. L'image n'est pas signée mais on veut quand même la télécharger
3. L'image est signée

Voici les 3 cas en pratique :

1. Image non-signée :

```
> docker pull armv7/armhf-debian
Using default tag: latest
Error: remote trust data does not exist for docker.io/armv7/armhf-debian:
↪ notary.docker.io does not have trust data for
↪ docker.io/armv7/armhf-debian
```

Quand on essaie de *pull* une image (mais c'est également le cas pour d'autres commandes comme `docker run`) et que celle-ci n'est pas signée, alors Docker indique qu'il n'a pas pu établir une confiance avec cette image.

Ici, il existe deux choix : soit on ignore ce message et on télécharge tout de même cette image, soit on cherche une autre image similaire qui a été signée.

2. Image non-signée, avertissement ignoré :

Pour quand même télécharger cette image, on peut utiliser le flag `-disable-content-trust` sur les commandes Docker concernées ou alors on peut redéfinir la variable d'environnement `DOCKER_CONTENT_TRUST` à 0.

```
> docker pull --disable-content-trust armv7/armhf-debian
Using default tag: latest
latest: Pulling from armv7/armhf-debian
cc738ad9b216: Pull complete
a3ed95caeb02: Pull complete
Digest:
 ↪  sha256:7c04a67927a012551a19d359f745480ae77afed229eb1ab0eac403b45fbf5f37
Status: Downloaded newer image for armv7/armhf-debian:latest
```

On note que pour lancer un container, il faut repréciser le flag sans quoi le container refusera de se lancer.

```
> docker run -it armv7/armhf-debian
docker: notary.docker.io does not have trust data for
 ↪  docker.io/armv7/armhf-debian.
See 'docker run --help'.
```

```
> docker run --disable-content-trust -it armv7/armhf-debian
root@28f7dca7a5f5:/#
```

3. Image signée Dans la pratique on se rend compte qu'il n'y a pas encore beaucoup d'images compatible ARM qui sont signées. D'ailleurs pour cet exemple, on `pull` une image signée x64 sur la cible. Il est évident que cette image ne va pas pouvoir être lancée mais c'est juste pour l'exemple.

```
> docker pull ubuntu
Using default tag: latest
Pull (1 of 1): ubuntu:latest@sha256:1bea66e185d[...]
sha256:1bea66e185d[...]: Pulling from library/ubuntu
203137e8afd5: Pull complete
2ff1bbbe9310: Pull complete
933ae2486129: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:1bea66e185d[...]
Status: Downloaded newer image for ubuntu@sha256:1bea66e185d[...]
Tagging ubuntu@sha256:1bea66e185d[...] as ubuntu:latest
```

Heureusement, il existe un moyen de garantir que l'intégrité de l'image même si celle-ci n'a pas été signée. La seule condition est qu'il faut connaître le *digest* (empreinte) de l'image.

```
> docker pull armhf/alpine@sha256:9f7b4923[...]
sha256:9f7b4923[...]: Pulling from armhf/alpine
06da76670c3f: Pull complete
Digest: sha256:9f7b4923[...]
Status: Downloaded newer image for armhf/alpine@sha256:9f7b4923[...]
```

Ici par exemple, on remarque que l'image `armhf/alpine:latest` n'est pas signée. Par contre, si on spécifie le *digest* de cette image, il est possible de la téléchargée sans avoir besoin d'indiquer le flag `-disable-content-trust`.

De plus, indiquer le *digest* dans le Dockerfile pour la commande `FROM xxx` est une bonne pratique car on s'assure de toujours avoir la même image qui est téléchargée et de ne pas avoir de surprise si l'image a été mise à jour.

On peut trouver ce *digest* quand on pull une image :

```
Digest: sha256:9f7b4923[...]
```

Ou avec la commande `docker images -no-trunc` :

```
> docker images --no-trunc
REPOSITORY          TAG      IMAGE ID            CREATED         SIZE
armhf/alpine        <none>   sha256:a9e1[...]    14 hours ago    3.906 MB
ubuntu              latest   sha256:9743[...]    9 weeks ago     187.9 MB
ubuntu              <none>   sha256:9743[...]    9 weeks ago     187.9 MB
armv7/armhf-ubuntu  latest   sha256:43b1[...]    7 months ago    130.5 MB
armv7/armhf-debian  latest   sha256:8d68[...]    8 months ago    104.9 MB
```

6.1.11 5.3 - Restrict Linux Kernel Capabilities within containers

Les containers sont lancés avec des permissions (*kernel capabilities*) par défaut. Ces permissions permettent d'effectuer des opérations qui demandent les droits root. Par exemple. la permission `net_bind_service` permet à un container de *binder* un port privilégié (ports inférieurs à 1024). Il est possible et recommandé de ne donner que les permissions qui sont strictement nécessaire à l'exécution du container. L'idée ici étant de ne donner aux containers qu'une partie des droits root, voire carrément aucun.

Il est par exemple inutile pour la plupart des containers d'avoir la permission `SYS_MODULE` qui permet charger et de décharger des modules kernel.

Les permissions accordées par défaut sont les suivantes[4] :
- CAP_CHOWN
- CAP_DAC_OVERRIDE
- CAP_FSETID
- CAP_FOWNER
- CAP_MKNOD
- CAP_NET_RAW
- CAP_SETGID
- CAP_SETUID
- CAP_SETFCAP
- CAP_SETPCAP
- CAP_NET_BIND_SERVICE
- CAP_SYS_CHROOT
- CAP_KILL
- CAP_AUDIT_WRITE

La liste complète des permissions et leur description se trouve ici : `http://man7.org/linux/man-pages/man7/capabilities.7.html`

Pour ajouter ou supprimer des permissions, on utilise les arguments `-cap-add` et `-cap-drop` de la commande `docker run`.

Voici un exemple. Ici on veut montrer ce qui se passe quand on retire la permission d'effectuer un `chroot`.

```
> docker run -it armhf/alpine /bin/sh
/ # ls
bin   etc   lib       media   proc   run    sys   usr
dev   home  linuxrc   mnt     root   sbin   tmp   var
/ # mkdir rootfs
/ # cd rootfs/
/rootfs # cp -R ../bin/ .
```

```
/rootfs # ls
bin   dev   etc   home   lib   linuxrc  media   mnt   proc
/rootfs # chroot .
/ # ls
bin   dev   etc   home   lib   linuxrc  media   mnt   proc
/ # exit
/rootfs # ls
bin   dev   etc   home   lib   linuxrc  media   mnt   proc
/rootfs # exit


> docker run --cap-drop="SYS_CHROOT" -it armhf/alpine /bin/sh
/ # mkdir rootfs
/ # cd rootfs/
/rootfs # cp -R ../bin .
/rootfs # chroot .
chroot: can't change root directory to '.': Operation not permitted
/rootfs #
```

Idéalement, il faudrait retirer toutes les permissions pour un container et n'ajouter que celles qui sont requises pour le bon fonctionnement du container. Pour ce faire, on peut utiliser l'argument `-cap-drop=all` pour retirer toutes les permissions et on utilise ensuite l'argument `-cap-add={"permission1", "permission2, "permissionN"}` pour ajouter les permissions requises.

6.1.12 5.4 - Do not use privileged containers

Le flag `-privileged` accorde toutes les permissions kernel à un container. Il ne faut pas utiliser ce flag mais il faut plutôt ajouter uniquement les permissions requises. Voir sous-section 6.1.11.

6.1.13 5.6 - Do not run ssh within containers

Pour communiquer avec les containers depuis l'extérieur, on serait tenter de démarrer un *daemon* ssh à l'intérieur de ceux-ci. Mais ceci comporte plusieurs inconvénients :

- Si on a plusieurs containers et que chacun d'eux démarre un *daemon* ssh, il faut s'assurer qu'ils soient à jour. Si on faille de ssh est découverte, il faut mettre à jour et reconstruire tous les containers.
- Il faut gérer les accès et politiques de sécurité pour chaque serveur ssh
- Il faut gérer le stockage des clés et des mots de passes pour chaque serveur ssh

Il est plus sûr de n'avoir qu'un serveur ssh sur la machine hôte et d'ensuite pouvoir exécuter des commandes sur le container voulu. Pour ce faire on utilise la commande `docker exec` :

```
docker exec <container_id> <command>
```

Les principales raisons pour lesquelles on voudrait se connecter en ssh sur un container sont les suivantes :

- Sauvegarde des données du container dans un autre endroit
- Vérification des logs
- Modification de la configuration d'un container
- Redémarrage d'un service

Les 3 premiers points peuvent être résolus en utilisant des volumes. Un volume est un dossier stocké sur la machine hôte qui est monté dans un dossier du container. Grâce à ce mécanisme, il est possible de persister des données générées par un container mais aussi de pouvoir partager ce volume et les données qu'il contient avec d'autres containers.

Dans l'exemple qui suit, on veut sauver les logs de deux containers. Pour ce faire, on va utiliser un volume qu'on appelle *logs*. Ce volume est utilisé par les deux containers pour stocker les logs qu'ils produisent.

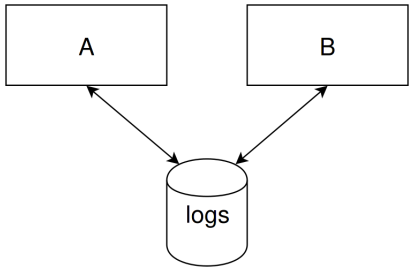


FIGURE 6.4 – Deux containers et un volume *logs*

Démarrage des deux containers (effectuer ce qui suit deux fois) et génération d'un faux log afin de montrer comment il est possible de récupérer ces logs. :

```
1  [alarm@alarm ~] > mkdir -p logs/XXX
2  [alarm@alarm ~] > docker run -v `pwd`/logs/XXX:/var/log -it armhf/alpine
   ↪ /bin/sh
3
4  / # echo "Error from container XXX..." > /var/log/error_`date
   ↪ +"%Y-%m-%d"`.log
5
6  [alarm@alarm ~]> cd logs/XXX/
7  [alarm@alarm XXX]> ls
8  error_2016-05-29.log
9  [alarm@alarm XXX]> cat error_2016-05-29.log
10 Error from container XXX...
```

Remarque : XXX est à remplacer par A et B, le nom des deux containers.

Ce volume peut être ensuite accédé en lecture seule par un container de log qui aurait pour but de faire remonter les logs sur un serveur de log ou dans le cloud par exemple.

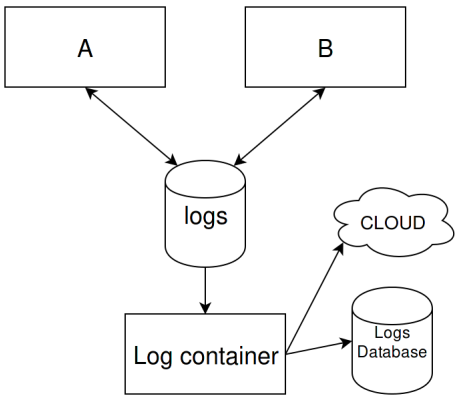


FIGURE 6.5 – Centralisation des logs et sauvegarde de ceux-ci

Pour le dernier cas, le redémarrage d'un service, il est possible d'ouvrir un terminal dans un container lancé grâce à la commande suivante :

```
docker exec -it <container-id> /bin/sh
```

Remarque : Le terminal sera lancé avec le même utilisateur que celui précisé au lancement du container. Cet utilisateur n'a peut être pas les droits de gérer les services (ce qui serait légitime). Dans ce cas, on peut toujours envoyer des signals avec la commande `docker kill -s <signal>` ou se connecter au socket démarré par le service pour lui demander de redémarrer. Les détails de cette technique sont expliqués ici : `https://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker/`. L'autre idée est de concevoir son container de manière à pouvoir le détruire et en recréer un autre sans que cela gêne le fonctionnement général. En théorie, cette solution est tout à fait viable car les containers Docker sont conçus pour être isolés et indépendants. A partir du moment où les données ne sont pas stockées uniquement dans le container, il ne devrait pas y avoir de problèmes.

6.1.14 5.10 - Limit memory usage for container

Il est possible et recommandé de limiter les ressources offertes à un container. Ces ressources peuvent être entre autres la mémoire vive ou l'utilisation CPU (voir section sous section suivante).

Limiter les ressources peut être une nécessité afin d'éviter un débordement d'un container. Si celui-ci se met à consommer toutes les ressources car il est victime d'une attaque (déni de service par exemple) ou d'un bug, le système risque de se geler et l'on devra être obligé de le redémarrer. Ceci est d'autant plus problématique si d'autres containers étaient lancés. En limitant les ressources d'un container, celui-ci pourrait dysfonctionner sans gêner les autres processus (ou containers) de la machine hôte.

Pour mettre en place cette limitation, il suffit d'ajouter le flag `-memory`. Dans l'exemple qui suit, on limite la mémoire à 100 MB.

```
docker run -it --memory 100m armhf/alpine /bin/sh
```

Voici un exemple plus complet fortement inspiré de ce site `https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/`. On utilise un programme appelé `stress` qui permet, on le devine, de stresser une machine. Voici le Dockerfile permettant de créer l'image `docker-test-memory` que l'on va utiliser.

```
FROM armhf/ubuntu

MAINTAINER "Gary Marigliano" <gary.marigliano@master.hes-so.ch>

RUN apt-get update && apt-get install -y stress

ENTRYPOINT ["stress"]
```

On construit l'image :

```
docker build -t docker-test-memory .
```

On va maintenant "stresser" avec 128 MB sans restriction sur la mémoire :

```
docker run -d docker-test-memory --vm 1 --vm-bytes 128M --vm-hang 0
```

On peut observer la consommation mémoire du container avec la commande `docker stats <container_id>` :

```
CONTAINER           CPU %   MEM USAGE/LIMIT       MEM %   NET I/O
a33c026010a1        0.00%   128.1 MiB/846.6 MiB   15.13%  594 B/468 B
```

Si on lance le même stress test mais avec une restriction de la mémoire :

```
docker kill a33c026010a1
docker run -d --memory 256M docker-test-memory --vm 1 --vm-bytes 128M
↪ --vm-hang 0
```

On voit désormais qu'une limite a été appliquée. Comme cette limite n'a pas été franchie, le container continue à s'exécuter.

```
CONTAINER           CPU %   MEM USAGE/LIMIT     MEM %   NET I/O
7797ab311f53        0.00%   128.1 MiB/256 MiB   50.05%  468 B/468 B
```

Désormais, on va dépasser volontairement la mémoire autorisée pour voir se qu'il se passe :

```
docker kill 7797ab311f53
docker run -d --memory 64M docker-test-memory --vm 1 --vm-bytes 128M
↪ --vm-hang 0
```

Résultat, le container crashe sans perturber le système hôte.

```
> docker ps -a
CONTAINER ID  IMAGE                COMMAND                CREATED       STATUS
5b6964334214  docker-test-memory  "stress --vm 1 --vm-  10 secs ago  Exited
↪  (1) 8 secs ago
```

Remarque : La limite indiquée par le flag `-memory` va appliquée à la taille de la RAM et **du SWAP**[29]. C'est à dire que si on possède un SWAP et qu'on fixe une limite à 128 MB, alors le container pourra consommer jusqu'à 256 MB de mémoire (128 MB (RAM) + 128 MB (SWAP, soit une fois la mémoire allouée pour la RAM). La cible a été configurée sans SWAP.

6.1.15 5.11 - Set container CPU priority appropriately

On l'a vu, il est possible de limiter les ressources d'un container Docker. Parmi, ces ressources, il y a évidement le processeur.

Il est possible de limiter l'utilisation de processeur de deux manières :

1. En définissant une priorité (*cpu shares*) à un container
2. En définissant le nombre de cores qu'un container peut utiliser

Pour simuler une charge processeur, l'image `docker-test-memory` va être utilisée comme c'était le cas dans la sous section précédente. En effet, `stress` permet de stresser le processeur sur un nombre de core à choix.

Plusieurs cas d'utilisation vont être simuler pour mesure l'effet des limitations imposées. Il y aura donc deux containers. Le premier avec une priorité basse (20%) appelé *low-prority* et un second, nommé *high-priority*, avec une priorité haute (80%).

Les cas d'utilisation suivants vont être analysés :

1. En utilisant tous les cores disponibles
   (a) *low-priority* seul
   (b) *high-priority* seul

   (c) *low-priority* et *high-priority* ensemble
2. En choisissant le(s) core(s) à utiliser
   (a) *low-priority* seul sur le core 0
   (b) *high-priority* seul sur le core 1
   (c) *low-priority* et *high-priority* ensemble sur le core 0

Remarque : la cible utilisée pour ces tests est un *ODROID C1* qui possède 4 cores. Ces tests sont reproductibles sur la cible originale "ODROID-XU3 Lite " sans soucis mais une autre carte a été choisie pour des questions pratiques. En effet, ma carte *ODROID C1* a été configurée pour être accessible depuis l'extérieur ce qui est bien pratique pour travailler dans le train.

Remarque : Entre chaque cas d'utilisation, on suppose que les containers ont été tuées avec la commande `docker kill <container_id>`.

CPU shares

On peut définir une priorité à un container en lui donnant un poids relatif/part (*share*). Une part peut varier entre 0 (poids minium → priorité minimale) et 1024 (poids maximum → priorité maximale). Par défaut, un container est lancé avec une part égale à 1024. Donc deux containers lancés occuperont le processeur à temps égal.

Remarque : On part de l'hypothèse que peu de processus tournent sur la cible au moment de l'exécution des commandes suivantes afin de ne pas perturber les mesures.

Cas d'utilisation - *low-priority*, tous les cores disponibles :

```
docker run --name=low-priority --cpu-shares=205 -d docker-test-memory --cpu
↪ `nproc`
```

Explication des paramètres :
- `-name` : nom du container
- `-cpu-shares` : nombre de parts, $205 \simeq 204.8 = 1024 \times 0.2$, 0.2 pour 20%
- `-cpu` : argument pour `stress`, indique le nombre de workers qui vont stresser la machine. Ici, on décide de mettre autant de workers que de cores sur la machine. On recupère ces cores grâce à la commande `nproc`.

Mesure de l'activité processeur :

```
> docker stats low-priority
low-priority   398.95%   188 KiB/846.6 MiB   0.02%   468 B/468 B
```

Comme on peut le constater, l'utilisation des processeurs est de presque... 400% ! $400 = 4 \times 100 =$ *nombre de cores* $\times$ *taux d'utilisation*. On voit donc que la cible est utilisée au maximum de ses capacités alors qu'on lui avait demandé de n'utiliser que 20% (`-cpu-shares=205`) de celles-ci. On explique cela par le fait qu'il s'agit d'un poids *relatif* et par conséquent si aucun autre processus (processus hôte ou container docker) n'est lancé, il n'y a aucune raison de ne pas profiter de l'intégralité des ressources.

Preuve si on lance un processus hôte :

```
> md5sum /dev/random & # md5sum est monocore
> docker stats low-priority
low-priority   296.71%   188 KiB/846.6 MiB   0.02%   594 B/468 B
```

Cas d'utilisation - *high-priority*, tous les cores disponibles :

```
> docker run --name=high-priority --cpu-shares=819 -d docker-test-memory
↪ --cpu `nproc`
> docker stats high-priority
high-priority   398.78%   196 KiB/846.6 MiB   0.02%  468 B/468 B
```

Avec cette fois 80% ($80 = 1024 \times 0.8$) de parts, on voit que le processeur reste pleinement utilisé pour la même raison évoquée précédemment ; si aucun autre processus (gourmand) est lancé, alors on utilise l'intégralité de la charge disponible.

Cas d'utilisation - *low-priority* et *high-priority* ensemble, tous les cores disponibles :

Lancement des deux containers :

```
docker run --name=low-priority  --cpu-shares=205 -d docker-test-memory
↪ --cpu `nproc` && \
docker run --name=high-priority --cpu-shares=819 -d docker-test-memory
↪ --cpu `nproc`
```

Utilisation du processeur :

```
CONTAINER      CPU %     MEM USAGE/LIMIT     MEM %   NET I/O
high-priority  319.41%   280 KiB/846.6 MiB   0.03%   768 B/468 B
low-priority   79.74%    532 KiB/846.6 MiB   0.06%   936 B/468 B
```

On le voit, *high-priority* occupe 80% de l'utilisation CPU ($320 = 400 * 0.8$) et *low-priority* occupe 20% ($80 = 400 * 0.2$).

Limitation du nombre de cores

TODO

Cas d'utilisation - *low-priority* seul sur le core 0 :

```
docker run --name=low-priority --cpu-shares=205 --cpuset-cpus=0 -d
↪  docker-test-memory --cpu `nproc`
```

Utilisation CPU :

```
> docker stats low-priority
low-priority   100.73%   356 KiB/846.6 MiB   0.04%   468 B/468 B
```

On voit que le container tourne sur un seul core (100% au lieu de 400%) malgré qu'il y ait toujours 4 workers (car `nproc` = 4). De nouveau, la priorité de 20% est ignorée car il n'y a qu'un seul container qui est exécuté actuellement.

Remarque : On garde ce container exécuté pour le prochain cas d'utilisation.

Cas d'utilisation - *high-priority* seul sur le core 1 :

```
docker run --name=high-priority --cpu-shares=819 --cpuset-cpus=1 -d
↪  docker-test-memory --cpu `nproc`
```

Utilisation CPU :

```
> docker stats low-priority high-priority
high-priority   100.14%   272 KiB/846.6 MiB   0.03%   468 B/468 B
low-priority    100.48%   356 KiB/846.6 MiB   0.04%   1.018 KiB/468 B
```

Comme on le remarque, *high-priority* se comporte de la même manière *low-priority* car ils sont "isolés" dans leur core respectif qu'ils utilisent pleinement.

Remarque : Ici aussi, on conserve les containers actuellement exécutés pour le dernier cas d'utilisation.

Cas d'utilisation - *low-priority* et *high-priority* ensemble sur le core 0 : On lance le container *high-priority* sur le core 0 désormais. On le nomme cette fois-ci ***high-priority-c0*** car il tourne sur le core 0.

```
docker run --name=high-priority-c0 --cpu-shares=819 --cpuset-cpus=0 -d
↪  docker-test-memory --cpu `nproc`
```

Utilisation CPU :

```
> docker stats low-priority high-priority high-priority-c0
high-priority       100.49%             192 KiB/846.6 MiB   0.02%
↪  866 B/468 B
high-priority-c0    79.25%              192 KiB/846.6 MiB   0.02%
↪  398 B/398 B
low-priority        20.15%              248 KiB/846.6 MiB   0.03%
↪  1.406 KiB/468 B
```

Remarque : Il est tout à fait possible d'exécuter un container sur plusieurs cores à la fois. Par exemple. il suffit d'utiliser le flag `-cpuset-cpus={0,2,3}` pour lancer le container sur les cores 0, 2 et 3.

6.1.16 Situation finale

TODO

6.2 Séparation des données Docker dans une partition chiffrée

Dans cette section, une partition chiffrée sera créée sur la cible afin d'isoler les données (images et containers principalement) de Docker du reste du système. La partition sera chiffrée afin d'accroître la sécurité des données.

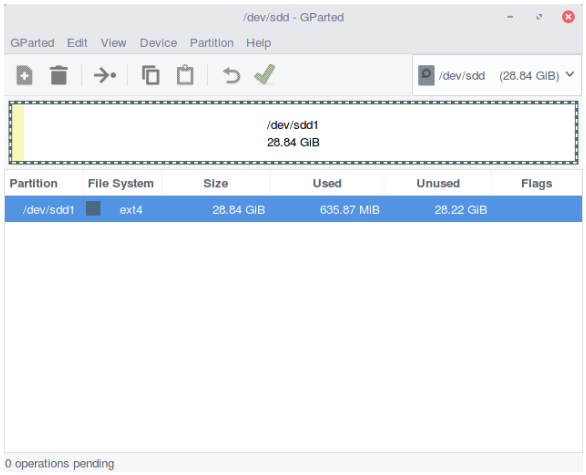
Les étapes suivantes seront réalisées :

1. Redimensionnement de la partition principale
2. Création d'une partition chiffrée
3. Montage manuel de la partition chiffrée
4. Montage au boot de la partition chiffrée
5. Preuve de bon fonctionnement du montage automatique
6. Binding des données de Docker sur la partition chiffrée
7. Analyse de la solution proposée

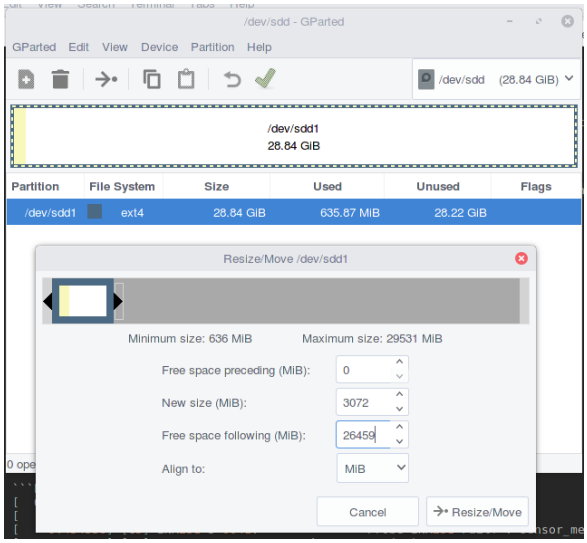
6.2.1 Redimensionnement de la partition principale

Actuellement, il n'y a qu'une partition sur la cible ; la partition principale. Il faut réduire la partition principale et créer une partition dans le nouvel espace libre. Pour ce faire, l'outil GParted [5] a été utilisé car il permet de voir visuellement ce qui est réalisé.

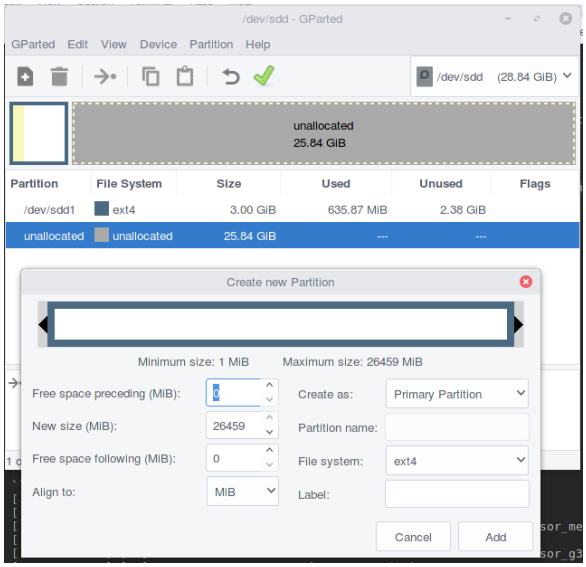
5. GParted : `http://gparted.org/`



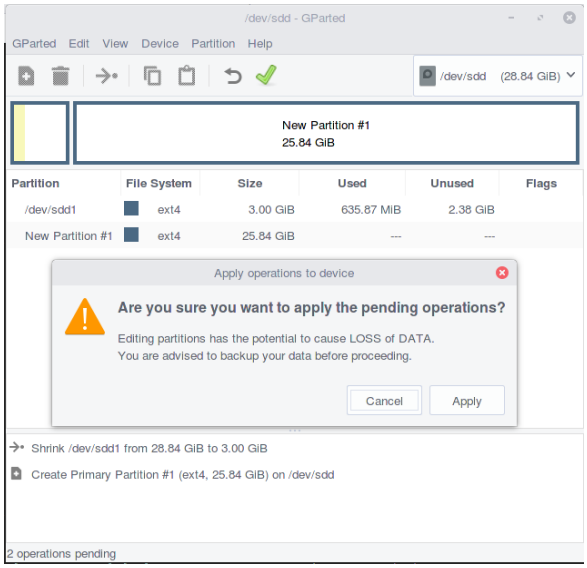
(a) Situation initiale - Une seule partition



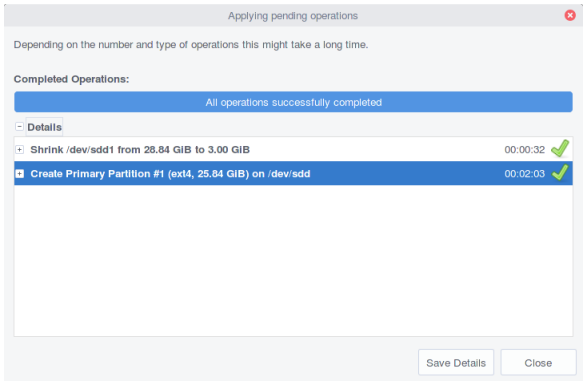
(b) Redimensionnement de la partition principale



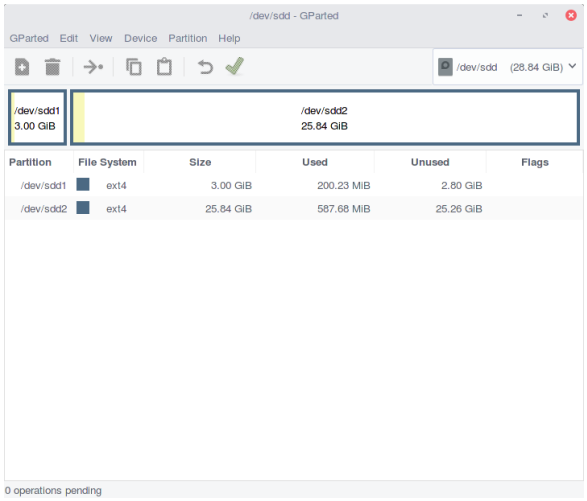
(c) Création de la partition pour les données Docker



(d) Confirmation



(e) Partitionnement en cours...

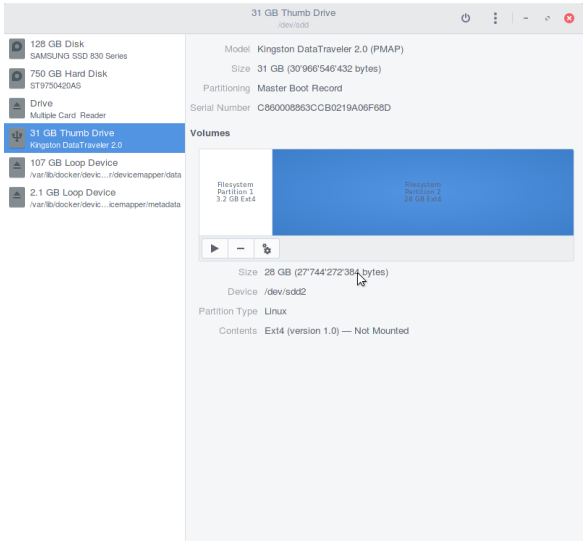


(f) Situation finale - Deux partitions, la principale et celle pour Docker

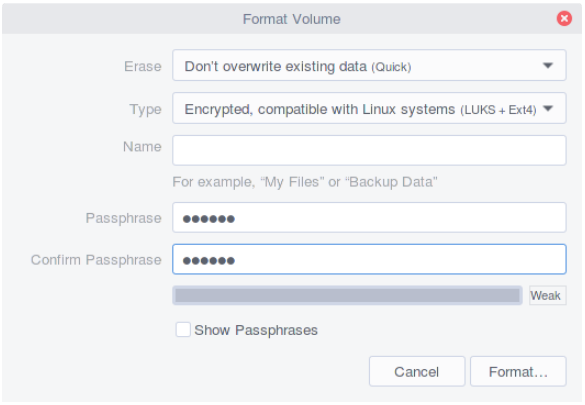
FIGURE 6.6 – Redimensionnement de la partition principale

6.2.2 Création d'une partition chiffrée

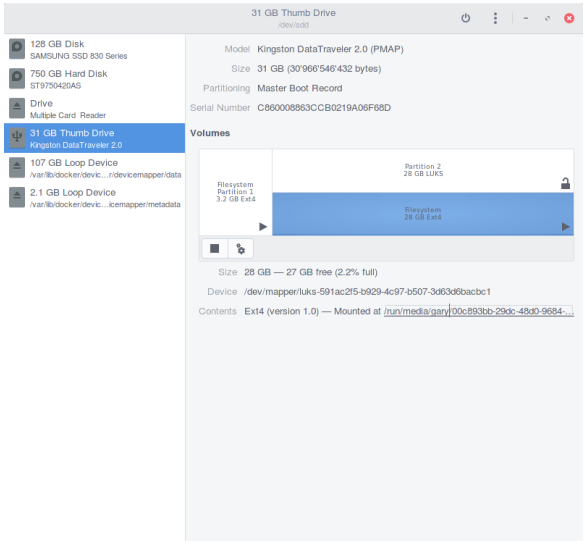
Maintenant que la carte SD correctement partitionnée, il faut créer la partition chiffrée avec LUKS. Pour ce faire, Gnome Disks Utility est utilisé [6].



(a) Sélection de la partition à chiffrer. Cliquer sur la roue crantée et sélectionner "Format Partition...".



(b) Sélectionner "Encrypted, compatible with Linux systems (LUKS + ext4)" et saisir une passphrase (mot de passe)



(c) La partition chiffrée est créée

FIGURE 6.7 – Création de la partition chiffrée avec Gnome Disk Utility

6.2.3 Montage manuel de la partition chiffrée

Maintenant que la partition chiffrée a été créée, on peut replacer la carte SD dans la cible et la démarrer.

Pour monter manuellement la partition, il faut réaliser les commandes suivantes :

```
1 sudo mkdir /mnt/docker-data
2 sudo cryptsetup luksOpen /dev/mmcblk0p2 docker-data
3 sudo mount /dev/mapper/docker-data /mnt/docker-data/
```

6. Gnome Disk Utility : `https://doc.ubuntu-fr.org/gnome-disk-utility`

`/dev/mmcblk0p2` représente la partition chiffrée de la carte SD.

6.2.4 Montage au boot de la partition chiffrée

Monter manuellement la partition n'est pas une solution possible si l'on souhaite y mettre les données de Docker dessus. En effet, pour que Docker stocke ces données (rappel : les images et les containers) dans la partition chiffrée, il faut monter cette partition au démarrage de la cible et créer un *bind mount* sur `/var/lib/docker`.

Avant d'effectuer le *bind mount*, il faut faire en sorte de monter la partition chiffrée au boot.

Pour monter la partition chiffrée au démarrage du système, il faut ouvrir le fichier `/etc/crypttab` sur la cible, et ajouter la ligne suivante :

```
docker-data /dev/mmcblk0p2  none luks
```

De manière similaire, il faut ajouter cette ligne dans le fichier `/etc/fstab` :

```
/dev/mapper/docker-data /mnt/docker-data ext4 rw 0 0
```

6.2.5 Preuve de bon fonctionnement du montage automatique

Pour vérifier que tout ceci fonctionne bien, il suffit de redémarrer la cible et on devrait observer les lignes suivantes au boot :

```
[  OK  ] Found device /dev/mmcblk0p2.
         Starting Cryptography Setup for docker-data...
Please enter passphrase for disk docker-data on /mnt/docker-data! ******
[***   ] (1 of 2) A start job is running for...for docker-data (13s / no
↪ limit)[   22.871246] [c5] NET: Registered pr8
[**    ] (1 of 2) A start job is running for...for docker-data (13s / no
↪ limit)[   23.925487] [c6] bio: create slab <1
[  OK  ] Found device /dev/mapper/docker-data.
[  OK  ] Started Cryptography Setup for docker-data.
[  OK  ] Reached target Encrypted Volumes.
         Mounting /mnt/docker-data...
[   24.117386] [c5] EXT4-fs (dm-0): mounted filesystem with ordered data
↪ mode. Opts: (null)
[  OK  ] Mounted /mnt/docker-data.
```

Le système va se bloquer pour que la saisie de la passphrase soit effectuée :

```
Please enter passphrase for disk docker-data on /mnt/docker-data! ******
```

Une fois connecté, il est tout à fait possible de d'écrire des données sur la partition chiffrée :

```
echo "LUKS je suis ton père !" > /mnt/docker-data/starwars.txt
```

6.2.6 Binding des données de Docker sur la partition chiffrée

Maintenant, on veut faire en sorte que les données de Docker soient dans cette partition chiffrée. Pour ce faire, on va effectuer un *bind mount* entre un dossier dans la partition chiffrée et le dossier de Docker.

Un *bind mount* n'est rien d'autre qu'un montage de dossier dans un autre dossier. On va donc monter `/mnt/docker-data/docker` dans `/var/lib/docker`. De cette manière, aucune configuration supplémentaire de Docker n'est nécessaire et ceci est donc complètement transparent pour lui. Cette opération est persistante après un redémarrage. Les *bind mount* sont également utilisés comme redirection pour ne pas avoir de points de montage introuvables s'ils ont été déplacés.

Attention : Les opérations suivantes impliquent la perte de toutes les images et containers précédemment créés.

Création du dossier `docker` sur la partition chiffrée, à faire sur la cible :

```
sudo systemctl stop docker # on stoppe le service Docker avant tout
cd /mnt/docker-data
mkdir -p var/lib/docker
sudo rm -rf /var/lib/docker/ # supprime toutes les données de Docker !!!
sudo mkdir /var/lib/docker
```

Sur la cible, on fait ensuite un *bind mount* entre la partition chiffrée et le dossier Docker de base

```
sudo vim /etc/fstab

# dans ce fichier, on ajoute la ligne suivante
/mnt/docker-data/var/lib/docker /var/lib/docker none bind 0 0
```

A la suite de ces modifications, le fichier /etc/fstab ressemble à ceci :

```
#
# /etc/fstab: static file system information
#
# <file system> <dir>   <type>  <options>       <dump>  <pass>
/dev/mapper/docker-data /mnt/docker-data ext4 rw 0 0
/mnt/docker-data/var/lib/docker /var/lib/docker none bind 0 0
```

Désormais les données de Docker sont stockées sur une partition séparée et chiffrée. Pour prouver que c'est bien le cas, il suffit de relancer le benchmark de sécurité :

```
cd /home/alarm/docker/docker-bench-security
sudo bash docker-bench-security.sh

# la sortie indique:
[PASS] 1.1  - Create a separate partition for containers
```

6.2.7 Analyse de la solution proposée

Stocker les données de Docker dans une partition chiffrée est une bonne initiative mais il faut garder à l'esprit les implications suivantes :

- La partition chiffrée est montée au boot. Ceci implique quand cas de reboot de la machine, il faut avoir un accès physique à celle-ci car le service ssh n'est pas encore lancé à ce moment là. En cas de dépannage à distance, il n'est pas donc possible de reboot la cible.
- Les performances quand on utilise Docker devraient être diminuées dû au chiffrement
- L'utilisation d'une partition séparée permet de réinstaller le système sans perdre les données liées à Docker et ses containers.
- Le chiffrement des containers Docker apporte une plus value en terme de sécurité

6.3 Conclusion sur la configuration du système d'exploitation hôte

TODO dire ce qui s'est passé et ce qui serait bien de faire en plus (project nautilus), GRSecurity, SELinux, AppArmor.

7. Création et utilisation des images Docker

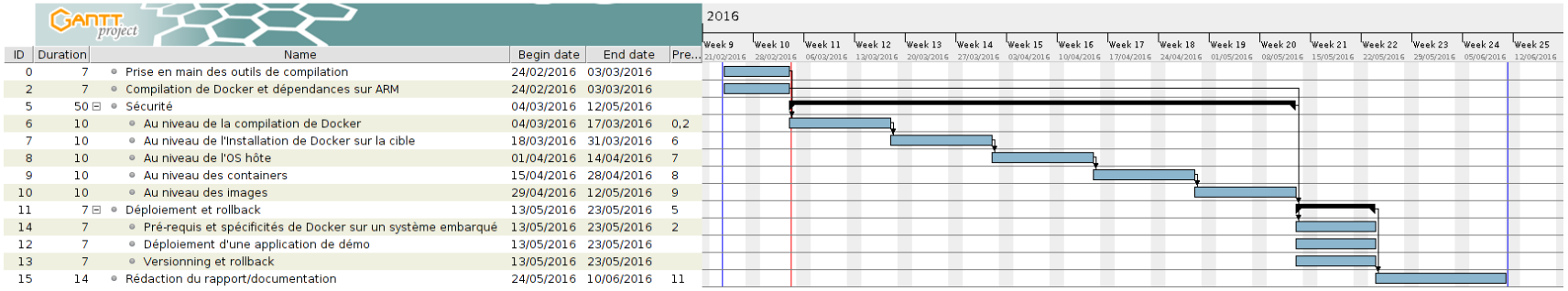
TODO : ce chapitre montre comment et pourquoi créer et utiliser des images Docker qui soient sécurisées et adaptées au monde de l'embarqué (taille principalement). Utilisation des hash cryptographiques pour l'OS utilisé et les versions des logiciels installés. Montrer un cas pratique avec une image commune 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

9.1 Planning initial



ID	Duration	Name	Begin date	End date	Pre...
0	7	Prise en main des outils de compilation	24/02/2016	03/03/2016	
2	7	Compilation de Docker et dépendances sur ARM	24/02/2016	03/03/2016	
5	50	Sécurité	04/03/2016	12/05/2016	
6	10	Au niveau de la compilation de Docker	04/03/2016	17/03/2016	0,2
7	10	Au niveau de l'Installation de Docker sur la cible	18/03/2016	31/03/2016	6
8	10	Au niveau de l'OS hôte	01/04/2016	14/04/2016	7
9	10	Au niveau des containers	15/04/2016	28/04/2016	8
10	10	Au niveau des images	29/04/2016	12/05/2016	9
11	7	Déploiement et rollback	13/05/2016	23/05/2016	5
14	7	Pré-requis et spécificités de Docker sur un système embarqué	13/05/2016	23/05/2016	2
12	7	Déploiement d'une application de démo	13/05/2016	23/05/2016	
13	7	Versionning et rollback	13/05/2016	23/05/2016	
15	14	Rédaction du rapport/documentation	24/05/2016	10/06/2016	11

FIGURE 9.1 – Planning initial - 04.03.2016

9.2 Planning final

img/initial_planning.png

FIGURE 9.2 – Planning final - TODO date

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)

Arrivé en fin de projet, les rapports de M. Bassang ont été relus afin de voir, si avec le recul, il y avait de nouvelles choses à en tirer. Après cette lecture, il a été jugé intéressant de faire une critique positive sur le travail effectué et les recommandations données dans ces rapports.

Le but ici est d'apporter un avis un peu moins informel, supplémentaire, complémentaire et plus récent sur l'utilisation de Docker en général.

Il faut insister sur le fait qu'il s'agit d'une critique positive qui a pour but d'améliorer le travail effectué. Avoir un feedback sur un projet est toujours bon à prendre.

11. Conclusion

TODO... ce qu'on voulait faire, ce qui a été fait, ce qu'il reste à faire/pourrait faire

Bibliographie

[1] 10.04 - Mount a LUKS partition at boot - Ask Ubuntu, mai 2016. http://askubuntu.com/questions/21025/mount-a-luks-partition-at-boot.

[2] Cgroups, mai 2016. https://fr.wikipedia.org/wiki/Cgroups.

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

[4] Default kernel capabilities, mai 2016. https://github.com/docker/docker/blob/master/oci/defaults_linux.go#L64-L79.

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

[6] Docker (software), mai 2016. https://en.wikipedia.org/wiki/Docker_%28software%29.

[7] Introduction to Linux Control Groups (Cgroups), mai 2016. https://sysadmincasts.com/episodes/14-introduction-to-linux-control-groups-cgroups.

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

[9] ODROID-XU3 Lite | Hardkernel, mai 2016. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141351880955.

[10] 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.

[11] Syslog, mai 2016. https://fr.wikipedia.org/wiki/Syslog.

[12] Ben Firshman. Faster and Better Image Distribution with Registry 2.0 and Engine 1.6, mai 2016. https://blog.docker.com/2015/04/faster-and-better-image-distribution-with-registry-2-0-and-engine-1-6/.

[13] Brand Pending. Shipping logs to rsyslog from a Docker container using the syslog Logging Driver, mai 2016. http://www.brandpending.com/blog/2015/11/9/shipping-logs-to-rsyslog-from-a-docker-container-using-the-syslog-logging-driver.

[14] Colin Barschel. Encrypt Partitions, mai 2016. http://sleepyhead.de/howto/?href=cryptpart.

[15] Diogo Mónica. Introducing Docker Content Trust, mai 2016. https://blog.docker.com/2015/08/content-trust-docker-1-8/.

[16] Docker Inc. API Documentation for Docker, mai 2016. https://docs.docker.com/engine/reference/api/docker_remote_api/.

[17] Docker Inc. Docker Documentation, mai 2016. https://docs.docker.com/.

[18] Docker Inc. Select a storage driver, mai 2016. https://docs.docker.com/engine/userguide/storagedriver/selectadriver/.

[19] Docker Inc. The daemon command description and usage, mai 2016. https://docs.docker.com/engine/reference/commandline/daemon/.

[20] 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.

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

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

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

[24] Jean-Roland Schuler. Filesystems - Secured Embedded Systems, mai 2016. Cours HES-SO Master.

[25] 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/.

[26] Jérôme Petazzoni. Deep dive into Docker storage drivers, mai 2016. https://jpetazzo.github.io/assets/2015-03-03-not-so-deep-dive-into-docker-storage-drivers.html.

[27] Loïc Bassang. Docker and Internet of Things. Technical report, Haute École d'ingénierie et d'architecture de Fribourg, 2015.

[28] Loïc Bassang. Docker and Internet of Things - Travail de Bachelor. Technical report, Haute École d'ingénierie et d'architecture de Fribourg, 2015.

[29] Marek Goldmann. Resource management in Docker, mai 2016. https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/.

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

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

[32] ram-0000. Présentation du protocole Syslog, mai 2016. http://ram-0000.developpez.com/tutoriels/reseau/Syslog/.

[33] Sehrope Sarkuni. Encrypting Docker containers on a Virtual Server, mai 2016. https://launchbylunch.com/posts/2014/Jan/13/encrypting-docker-on-digitalocean/.

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

[35] Viktor Petersson. Manage Docker resources with Cgroups, mai 2016. https://www.cloudsigma.com/manage-docker-resources-with-cgroups/.

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.