Virtualisation légère – TP n° 2

Construire des images Docker et leur sécurité

Pierre-Olivier *nemunaire* Mercier

Mercredi 21 septembre 2022

Abstract

Durant ce deuxième TP, nous allons voir comment créer nos propres images, et comment s'assurer qu'elles n'ont pas de vulnérabilités connues !

Sommaire

1 Contenir les applications pour éviter les fuites

Lorsque l'on gère un environnement de production, on souhaite bien évidemment éviter tout déni de service. Ou parfois, contenir un programme métier avec une fuite mémoire, dans certaines limites : il vaut parfois mieux le tuer et le relancer automatiquement, plutôt que d'attendre que potentiellement un autre processus se fasse tuer à sa place.

Pour cela, Docker expose tout un arsenal, reposant sur les cgroups du noyau Linux, que l'on verra plus en détail par la suite.

1.1 Limiter l'utilisation des ressources

1.1.1 Mémoire

Comme on peut s'y attendre, il est possible de limiter la mémoire que peut occuper un conteneur avec l'option `-m`/`--memory`.

1.1.2 CPU

En ce qui concerne la limitation d'utilisation du CPU, ce n'est pas aussi simple. En effet, on ne peut pas définir le nombre d'instructions par seconde qu'un conteneur est autorisé à consommer.

On ne peut définir qu'un taux d'utilisation relatif par rapport à l'ensemble du système (ou du groupe de processus auquel il appartient). Ce taux est appliqué par l'ordonnanceur, lorsqu'il détermine la prochaine tâche qui sera exécutée.

Ainsi, lorsque la machine n'est pas chargée, que le processeur n'a pas constamment du travail à effectuer, l'ordonnanceur ne va pas empêcher une tâche très consommatrice en puissance de calcul de s'exécuter.

Par contre, sous une forte charge, si l'on définit que notre conteneur exécutant un cpuburn ne peut pas utiliser plus de 50% des ressources de la machine, ce pourcentage ne pourra effectivement pas être dépassé, l'ordonnanceur privilégiant alors les autres processus du système.

Par défaut, le taux maximal (1024 = 100%) d'utilisation CPU est donné aux nouveaux conteneurs, on peut le réduire en utilisant l'option `-c`/`--cpu-shares` : 512 = 50%, par exemple.

1.2 Sécuriser l'exécution

En plus des dénis de service, on peut également vouloir se protéger contre des attaques provenant des conteneurs eux-mêmes. On n'est pas à l'abri d'une vulnérabilité dans un des services exécutés dans un conteneur. Plusieurs mécanismes sont mis en place pour accroître la difficulté du rebond.

1.2.1 Capabilities

Un certain nombre de capabilities Linux sont retirées par Docker au moment de l'exécution du conteneur, on peut utiliser les options `--cap-add` et `--cap-drop` pour respectivement ajouter et retirer une capabilities.

Notez que l'option `--privileged` ne retire aucune capabilities à l'exécution.

Nous verrons plus tard de quoi sont capables ces *capabilities* exactement.

1.2.2 seccomp

Si les capabilities sont un regroupement grossier de fonctionnalités du noyau, seccomp est un filtre que l'on peut définir pour chaque appel système. Liste blanche, liste noire, tout est possible.

Docker filtre notamment tous les appels systèmes qui pourraient déborder à l'extérieur du conteneur : il n'est par exemple pas possible de changer l'heure dans un conteneur, car il n'y a aujourd'hui aucun mécanisme pour isoler les visions des dates d'un conteneur à l'autre.

Voici par exemple un fichier de profil seccomp, interdisant l'utilisation de l'appel système `nanosleep(2)` (utilisé notamment par `sleep(1)`) :

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
            "names": [
                "nanosleep"
            ],
            "action": "SCMP_ACT_ERRNO",
            "args": [],
            "comment": "",
            "includes": {},
            "excludes": {}
        }
    ]
}
```

On peut ensuite l'appliquer à un conteneur Docker :

```
42sh$ docker run -it --security-opt seccomp=nanosleep.json ubuntu /bin/bash
(cntnr)$ sleep 42
sleep: cannot read realtime clock: Operation not permitted
```

2 Construire des images

Jusqu'à maintenant, nous avons profité des images présentes sur les registres pour utiliser Docker. Sur ces registres, on trouve d'ailleurs non seulement des images officielles proposées directement par les éditeurs (`nginx`, `mysql`, ...), mais aussi des images conçues par les utilisateurs pour leurs propres besoins.

Et si nous aussi, nous construisions nos propres images ? Ça vous dit ?

2.1 Modification interactive

Pour créer une image, commençons par entrer dans un nouveau conteneur :

```
docker container run -it ubuntu /bin/bash
```

Nous voilà maintenant dans le conteneur ! Il est assez épuré, il n'y a rien de superflu : même pas d'éditeur de texte : ni vim, ni emacs, même pas `vi` !

La première chose à faire est de télécharger la liste des paquets. En effet, afin de ne pas livrer de superflu, la liste des paquets et son cache ne sont pas inclus dans le conteneur.

```
apt-get update
```

Il peut arriver que des paquets présents dans l'image ne soient pas à jour. De manière générale, il n'est pas recommandé de faire de mises à jour automatiques et systématiques des éléments présents dans l'image, à l'exception des mises à jour de sécurité[1]. En effet, une mise à jour qui apporte des changements peut altérer le comportement du conteneur, en fonction de la date à laquelle on le construit. Car on ne sait pas d'avance quelles versions de nos dépendances on va récupérer.

Si vous souhaitez disposer d'une nouvelle version de l'image, il est plutôt recommandé de contacter le mainteneur de l'image pour qu'il la mette à jour, en utilisant un nouveau tag s'il le juge nécessaire. Si cette solution n'est pas envisageable, alors il vaut mieux créer votre propre image, à partir de l'image de base : vous serez alors vous-même responsable de la bonne continuité de construction des images issues de votre image, sans que cela soit hasardeux au moment de la construction.

La liste des paquets récupérés, installons maintenant un programme : notre première image pourrait contenir notre éditeur de texte favori :

```
apt-get install nano
```

Lorsque l'installation de `nano` est terminée, quittons l'image en tapant `exit`.

Nous allons sauvegarder nos modifications en tant que nouvelle image Docker, avec la commande `commit` :

```
42sh$ docker ps
CONTAINER ID  IMAGE   COMMAND  STATUS      NAMES
91d17871d730  ubuntu  "bash"   Exited (0)  musing_tu

docker container commit 91d17871d730 my_nano
```

[1] Voir cet article : https://pythonspeed.com/articles/security-updates-in-docker/

en remplaçant `91d17871d730` par le nom ou l'identifiant du container qui doit servir de modèle. `my_nano` est le nom que vous voudrez utiliser à la place d'`ubuntu`.

L'action de *commit*, malgré le fait qu'elle crée une nouvelle image est très rapide : il se trouve que seules les différences avec l'image parente sont packagées. Les images sont en fait composées de couches : empilant les différences depuis le système de base !

2.1.1 À propos des couches

Revenons quelque-peu en arrière : lorsque nous avons fait notre premier `docker run hello-world`, rappelez-vous, Docker a téléchargé l'image en nous affichant la progression, juste avant de lancer le conteneur.

Analysons ensemble ces quelques lignes pour mieux comprendre de quoi les images se composent. Nous allons pour cela utiliser la commande `pull` pour récupérer une nouvelle image :

```
42sh$ docker image pull python:3
3: Pulling from library/python
23858da423a6: Pull complete
326f452ade5c: Pull complete
a42821cd14fb: Pull complete
8471b75885ef: Pull complete
8ffa7aaef404: Pull complete
15132af73342: Pull complete
aaf3b07565c2: Pull complete
736f7bc16867: Pull complete
94da21e53a5b: Pull complete
Digest: sha256:e9c35537103a2801a30b15a77d4a56b35532c964489b125ec1ff24f3d5b53409
Status: Downloaded newer image for python:3
docker.io/library/python:3
```

On remarque que plusieurs téléchargement ont lieu, chacun associé à un identifiant particulier. Une image est généralement découpée en plusieurs éléments. On parle en fait de *couches* puisqu'on les empile, dans un ordre précis.

Les couches sont une astuce formidable pour optimiser tant le téléchargement, l'espace de stockage nécessaire au cache d'images, que la création des conteneurs. De nombreux conteneurs vont utiliser les mêmes images de base : `debian`, `ubuntu`, `alpine`, ... il serait futile de systématiquement récupérer et stocker autant de systèmes de fichiers de base que d'images. Avec les couches, si deux images partagent la même version du système de fichiers de base, il ne sera téléchargé qu'une seule fois. On pourait le schématiser ainsi :

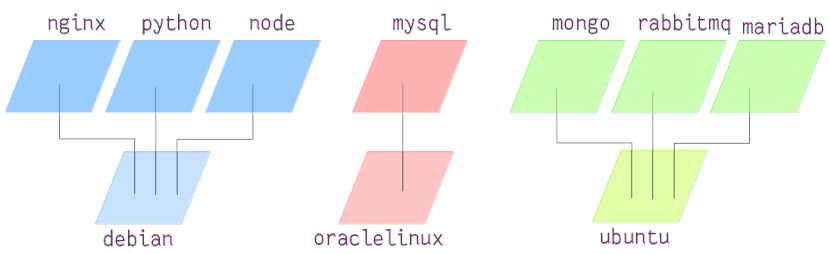

Figure 1: L'héritage des principales images officielles

Dans les faits, cela va même encore plus loin car Docker crée de nombreuses couches intermédiaires,

chacune peut être l'occasion d'une bifurcation.

Chaque couche est en fait un différentiel des dossiers et fichiers qui sont ajoutés, modifiés ou supprimés par rapport à la couche précédente.

Comment supprimer les couches d'images que je n'utilise plus ? Docker gère lui-même les couches, vous n'avez pas à vous en préoccuper. Si une image est mise à jour ou supprimée, toutes les couches rendues inutiles seront automatiquement supprimées.

Revenons au *commit* que nous avons fait précédemment. Nous avons ajouté `nano` par-dessus une image `ubuntu`. Naturellement, voici ce qu'il s'est passé :

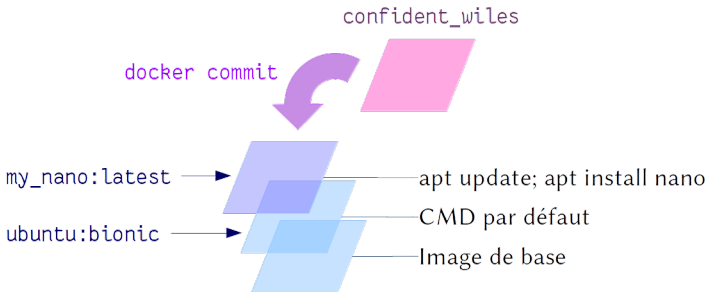

Figure 2: `docker commit`

Testons alors sans plus attendre notre nouvelle image :

```
docker container run -it my_nano /bin/bash
```

Vous constatez cette fois que vous pouvez lancer `nano` !

Nous avons donc créé une couche, elle contient juste le différentiel des fichiers ajoutés, à savoir le binaire `nano` et sa configuration par défaut (mais aussi le cache du gestionnaire de paquets `apt`).

Voyons maintenant comment automatiser cela.

2.1.2 Scripté ?

On peut automatiser les étapes ci-dessus avec un script qui ressemblerait à ça :

```
docker container run ubuntu apt-get update
docker container commit $(docker container ls -lq) my_nano_step-1
docker container run my_nano_step-1 apt-get install nano
docker container commit $(docker container ls -lq) my_nano
```

On obtiendra de la même manière notre image `my_nano` :

```
docker container run -it my_nano /bin/bash
```

contenant notre éditeur de texte favori.

On ne va pas réaliser ce script ni l'étoffer, car il existe justement un mécanisme de construction d'image : le `Dockerfile`.

2.2 Ma première image ... par `Dockerfile`

Pour construire une image, nous ne sommes pas obligés de passer par une série de *commits*. Docker dispose d'un mécanisme permettant d'automatiser la construction de nouvelles images. Nous pouvons arriver au même résultat que ce que l'on a réussi à faire précédemment en utilisant le `Dockerfile` suivant :

```
FROM ubuntu:jammy

RUN apt-get update
RUN apt-get install -y nano
```

La syntaxe d'un `Dockerfile` est simple : le premier mot de chaque ligne est l'intitulé d'une instruction (que l'on écrit généralement en majuscule), elle est suivie de ses arguments.

Dans notre exemple, nous utilisons `FROM` qui indique une image de départ à utiliser ; `RUN` est une commande qui sera exécutée dans le conteneur intermédiaire, dans le but de construire l'image. De la même manière que les `docker container run` de la partie précédente.

Vous avez remarqué que la première instruction que l'on utilise est `FROM`. Chaque image construite par un `Dockerfile` doit dépendre d'une autre image. Ici nous avons choisi de partir de l'image `ubuntu`.

Pour lancer la construction de la nouvelle image, créons un nouveau dossier ne contenant que notre fichier `Dockerfile`, plaçons-nous ensuite dedans, puis lançons la commande `build` :

```
docker image build --tag=my_editor .
```

On utilise l'option `--tag` pour donner un nom et un tag à l'image qui résultera de l'exécution de cette construction.

Attention de ne pas oublier le point à la fin de la commande ! Vous n'êtes plus sans savoir que Docker se compose d'un client et d'un serveur. Et c'est la partie serveur qui va s'occuper de construire l'image.
Le client transmet donc tout le contexte autour du Dockerfile (les fichiers, dossiers, sons-dossiers) à partir du chemin qu'on lui indique en dernier argument. Le point représente donc ici simplement le dossier courant. Tous les fichiers et dossiers présents ici seront transmis au daemon.

Une fois la construction de l'image terminée, nous pouvons la lancer et constater l'existence de notre éditeur favori :

```
docker container run -it my_editor /bin/bash
(in_cntr)# nano
```

2.2.1 `RUN` dans le `Dockerfile`

Dans un `Dockerfile`, chaque ligne est exécutée indépendamment des autres et correspondra à une nouvelle couche de notre image. Exactement comme on a réalisé le script à la fin de la partie précédente.

Cela signifie que l'exemple suivant **ne fonctionne pas** :

```
COPY db.sql /db.sql
RUN service mysqld start
RUN mysql -u root -p toor virli < /db.sql
```

Cet exemple ne fonctionne pas car le serveur MySQL est bien lancé dans le premier `RUN`, mais il se trouve brutalement arrêté dès lors que la commande `service` se termine. En fait, à chaque instruction, Docker réalise automatiquement l'équivalent un `docker run` suivi d'un `commit`. Et vous pouvez constater par vous-même que, en créant l'image `tinysql` à partir d'un simple `apt install mysql` :

```
docker container run tinysql service mysqld start
```

rend la main directement, sans laisser de `mysqld` dans l'arborescence de processus.

Pour avoir le résultat escompté, il faut exécuter les commandes ensemble :

```
COPY db.sql /db.sql
RUN service mysqld start && mysql -u root -p toor virli < /db.sql
```

Après le `RUN`, MySQL sera de nouveau tué, mais la seconde commande aura entre-temps pu ajouter des données.

En aucun cas, une commande exécutée par un `RUN` se retrouvera en cours d'exécution lorsque l'on invoquera un conteneur par `docker container run`. Seul la commande fournie par l'utilisateur ou la commande par défaut de l'image sera exécutée au lancement d'un conteneur.

2.2.2 Exposer des ports

Construisons maintenant un conteneur avec un service web :

```
FROM my_editor

RUN apt-get update
RUN apt-get install -y nginx

EXPOSE 80
```

L'instruction `EXPOSE` sera traitée plus tard par le client Docker (équivalent à l'argument `--expose`). Il s'agit d'une métadonnée qui sera attachée à l'image (et à toutes ses images filles). Elle ne crée d'ailleurs pas de couche supplémentaire dans notre image.

En précisant tous les ports qu'expose une image dans ses métadonnées, ces ports seront automatiquement exposés en utilisant l'option `-P` du `run` : cela assigne une redirection de port aléatoire sur la machine hôte vers votre conteneur :

```
42sh$ docker image build --tag=my_webserver .
42sh$ docker container run -it -P my_webserver /bin/bash
(cntnr)# service nginx start
```

Dans un autre terminal, lançons un `docker container ls`, pour consulter la colonne *PORTS* afin de connaître le port choisi par Docker pour effectuer la redirection.

Rendez-vous ensuite dans votre navigateur sur http://localhost:49153/.

2.2.3 Copier des fichiers dans l'image

Une autre action très courante est de vouloir recopier un fichier ou un binaire dans notre image : un fichier de configuration, un produit de compilation, des scripts pour contrôler l'exécution, ...

On va utiliser pour cela l'instruction `COPY` :

```
COPY myconfig.conf /etc/nginx/conf.d/my.conf
```

Cette instruction permet également de copier l'arborescence d'un dossier :

```
COPY myconfs/ etc/nginx/conf.d/
COPY mywebsite /usr/share/nginx/html/
```

Le comportement de la copie de dossier est différente du comportement que l'on a l'habitude d'avoir avec `cp -r`. Si la source du `COPY` est un dossier, c'est son contenu qui sera recopié récursivement, habituellement avec `cp` le dossier recopié puis son contenu.
Pour obtenir le même comportement, il faut bien indiquer une cible incluant le nom du dossier :

```
COPY docker-entrypoint.d /docker-entrypoint.d
```

Le dossier sera créé s'il n'existe pas, et le contenu du dossier source ser recopié.

À vous de jouer Utilisez l'instruction `COPY` pour afficher votre propre `index.html` remplaçant celui installé de base par `nginx`.

2.2.4 Les caches

Nous avons vu que chaque instruction de notre `Dockerfile` est exécutée dans un conteneur, qui génère une image intermédiaire. Cette image intermédiaire sert ensuite d'image de base pour le conteneur qui sera lancé avec l'instruction suivante.

Lorsqu'on lance la reconstruction du même `Dockerfile`, les images intermédiaires sont réutilisées, comme un cache d'instructions. Cela permet de gagner du temps sur les étapes qui n'ont pas changé. Ainsi, lorsque vous modifiez une instruction dans votre `Dockerfile`, les instructions précédentes ne sont pas réexécutées mais sont ressorties du cache.

Le cache se base principalement sur le contenu de chaque instruction du `Dockerfile` (pour les `COPY` et `ADD`, il va aussi regarder la date de dernière modification de fichier à copier ou à ajouter). Donc tant qu'une instruction n'est pas modifiée dans le `Dockerfile`, le cache sera utilisé.

Il est possible de ne pas utiliser le cache et de relancer toutes les étapes du `Dockerfile` en ajoutant l'option `--no-cache` au moment du `docker image build`.

Les couches du cache peuvent être partagées entre plusieurs conteneurs, c'est ainsi que vous pouvez partager facilement une plus grosse partie du système de fichiers.

Pour profiter au mieux du cache, on place les instructions qui sont le moins susceptibles de changer en haut du `Dockerfile`, celles qui changent le plus régulièrement à la fin. Ainsi, lorsqu'une reconstruction de l'image sera nécessaire, on gagnera du temps puisque le cache sera utilisé jusqu'à la première instruction changeante. Un `Dockerfile` bien ordonné peu facilement faire gagner de nombreuses minutes à ses utilisateurs.

?

Quelle place cela prend-t-il sur mon disque ? Nous pouvons afficher la taille de chaque image via la commande `docker image ls` :

```
42sh$ docker image ls
REPOSITORY        TAG     IMAGE ID       CREATED      SIZE
nginx             latest  2d389e545974   6 days ago   142MB
debian            stable  9b4953ae981c   7 days ago   124MB
nemunaire/youp0m  latest  2c06880e48aa   12 days ago  25MB
```

Si vous avez beaucoup d'images, cela peut paraître beaucoup, mais rappelez-vous que les images sont composées de couches qui sont souvent partagées entre plusieurs conteneurs.
Si on regarde l'espace vraiment utilisé, il est moindre :

```
42sh$ docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          3         3         167MB     0B
Containers      0         0         0B        0B
Local Volumes   0         0         0B        0B
Build Cache     0         0         0B        0B
```

Les couches partagées sont un gain non négligeable pour l'espace de stockage !

Par exemple, prenons le `Dockerfile` suivait :

```
FROM python:3.10
COPY build /usr/lib/python/grapher
EXPOSE 8080
RUN pip install pillow pygal
```

Il y a de fortes chances pour que vous travailliez sur le code de l'application, le dossier `build` sera donc très souvent mis à jour, alors que les dépendances ne bougeront sans doute plus …

Avec un tel `Dockerfile`, dès que le dossier `build` sera mis à jour les dépendances seront à nouveau téléchargées, puisque toutes les couches suivant la première qui change sont invalidées.

Une approche plus optimale serait donc de faire la `COPY` en dernier, car c'est l'opération qui changera le plus souvent. L'idéal étant que 90 % des reconstructions ne refassent que la dernière instruction, toutes les autres devraient être récupérées du cache.

2.2.5 Métadonnées pures

L'instruction `LABEL` permet d'ajouter une métadonnée à une image, sous forme de clef/valeur.

Une métadonnée courante[2] est d'indiquer le nom du mainteneur de l'image :

```
LABEL maintainer="Pierre-Olivier Mercier <nemunaire@nemunai.re>"
```

Dans notre `Dockerfile`, indiquez juste après l'image de base, vos noms, prénoms et mails de contact avec l'instruction `LABEL maintainer`, pour indiquer que c'est vous qui maintenez cette image, si des utilisateurs ont besoin de vous avertir pour le mettre à jour ou s'ils rencontrent des difficultés par exemple.

On le place dès le début, car comme c'est une information qui n'est pas amenée à changer, elle sera toujours retrouvée en cache.

2.2.6 Commande par défaut

Vous pouvez placer dans un `Dockerfile` une instruction `CMD` qui sera exécutée si aucune commande n'est passée lors du `run`, par exemple :

```
CMD nginx -g "daemon off;"
```

```
42sh$ docker image build --tag=my_nginx .
42sh$ docker container run -d -P my_nginx
```

L'option `-d` passée au `run` lance le conteneur en tâche de fond. Si vous constatez via un `docker container ls` que le conteneur s'arrête directement, retirez cette option pour voir ce qui ne va pas, ou utilisez la commande `docker container logs`.

Comme les `LABEL`, ce n'est pas une instruction qui change régulièrement. On la place plutôt au début du `Dockerfile`.

2.2.7 Construire son application au moment de la construction du conteneur ?

Comment faire lorsque l'on a besoin de compiler une application avant de l'intégrer dans le conteneur ?

On peut vouloir lancer la compilation sur notre machine, mais cela ne sera pas très reproductible et cela aura nécessité d'installer le compilateur et les outils liés au langage que l'on souhaite compiler. Peut-être que plusieurs versions de ces outils existent, laquelle choisir ? … Ok c'est trop compliqué.

D'un autre côté, si l'on fait cela dans un conteneur, celui-ci contiendra dans ses couches des données inutiles à l'exécution : les sources, les produits intermédiaires de compilation, le compilateur, n'ont rien à faire dans les couches de notre image.

Le meilleur des deux mondes se trouve dans les *Multi-stage builds* : au sein du même `Dockerfile`, on va réaliser les opérations de préparation dans un ou plusieurs conteneurs, avant d'agréger le contenu compilé au sein du conteneur final :

```
FROM gcc:4.9
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN gcc -static -static-libgcc -o hello hello.c

FROM scratch
COPY --from=0 /usr/src/myapp/hello /hello
CMD ["/hello"]
```

[2] Voir par exemple : https://github.com/nginxinc/docker-nginx/blob/master/stable/debian/Dockerfile#L8

Dans cet exemple, deux images distinctes sont créées : la première à partir de l'image gcc, elle contient tout le nécessaire pour compiler notre hello.c. Mais l'image finale (le dernier FROM de notre Dockerfile) est l'image vide, dans laquelle nous recopions simplement le produit de notre compilation.

L'image ainsi générée est minime, car elle ne contient rien d'autre que le strict nécessaire pour s'exécuter.

2.2.7.1 Étapes nommées

Nous avons utilisé --from=0 pour désigner la première image de notre Dockerfile. Lorsque l'on réalise des montages plus complexes, on peut vouloir donner des noms à chaque image, plutôt que de devoir jongler avec les numéros. Dans ce cas, on indiquera :

```
FROM gcc:4.9 as builder
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN gcc -static -static-libgcc -o hello hello.c

FROM scratch
COPY --from=builder /usr/src/myapp/hello /hello
CMD ["/hello"]
```

Par défaut la dernière étape du Dockerfile est retenue comme étant l'image que l'on souhaite tagger, mais il est possible de préciser quelle image spécifiquement on souhaite construire avec l'option --target :

```
42sh$ docker build --target builder -t hello-builder .
```

Cela peut être particulièrement utile si l'on dispose d'une image de debug, incluant tous les symboles, et une image de production, plus propre. On sélectionnera ainsi avec l'option --target l'un ou l'autre en fonction de l'environnement dans lequel on souhaite se déployer.

2.2.8 Déclarer des volumes

Tout comme nous pouvons déclarer préalablement dans le Dockerfile les ports qui sont normalement exposés par le conteneur, nous pouvons déclarer les volumes. L'instruction pour cela est VOLUME.

Il convient de l'utiliser pour déclarer les emplacements qui vont par défaut contenir des données à faire persister. Ce serait le cas de /var/lib/mysql pour les conteneurs MariaDB ou MySQL, /images/ pour notre image youp0m ...

2.2.9 D'autres instructions ?

Nous avons fait le tour des principales instructions et de leurs différents usages *classiques*. Il existe quelques autres instructions que nous n'avons pas présentées ici, pour aller plus loin, consultez la référence sur :
https://docs.docker.com/engine/reference/builder/

Pour mettre en application tout ce que nous venons de voir, réalisons le Dockerfile du service web youp0m que nous avons déjà utilisé précédemment.

Pour réaliser ce genre de contribution, on ajoute généralement un Dockerfile à la racine du dépôt.

Vous pouvez cloner le dépôt de sources de youp0m à :
https://git.nemunai.re/nemunaire/youp0m.git

Pour compiler le projet, vous pouvez utiliser dans votre `Dockerfile`

```
FROM golang:1.18
COPY . /go/src/git.nemunai.re/youp0m
WORKDIR /go/src/git.nemunai.re/youp0m
RUN go build -tags dev -v
```

Remarquez la puissance de Docker : vous n'avez sans doute pas de compilateur Go installé sur votre machine, et pourtant, en quelques minutes et à partir du seul code source de l'application et d'un `Dockerfile`, vous avez pu compiler sur votre poste le binaire attendu. WOW, non ?

2.3 Les bonnes pratiques

Pour chaque bonne pratique ci-dessous, vérifiez que vous la respectez bien, faites les modifications nécessaires dans votre `Dockerfile`.

2.3.1 Utilisez le fichier `.dockerignore`

Dans la plupart des cas, vos `Dockerfile` seront dans des dossiers contenant beaucoup de fichiers qui ne sont pas nécessaires à la construction de votre conteneur (par exemple, vous pouvez avoir un `Dockerfile` placé à la racine d'un dépôt git).

Afin d'améliorer les performances lors de la construction, vous pouvez exclure les fichiers et dossiers inutiles au conteneur en ajoutant un fichier `.dockerignore` dans le répertoire de votre `Dockerfile`.

Vous pouvez exclure les produits intermédiaires de compilation (`*.o`, ...) si vous utilisez un langage compilé, excluez également les produits de compilation si votre image construit le binaire. Dans le cas de NodeJS, vous allez sans doute vouloir exclure le dossier `node_modules` et faire un `npm install` dans votre `Dockerfile`. Cela permettra au passage de s'assurer que toutes les dépendances ont bien été enregistrées.

Ce fichier fonctionne de la même manière que le `.gitignore` : vous pouvez utiliser du globing.

Pour plus d'informations, vous pouvez consulter la documentation accessible à
https://docs.docker.com/engine/reference/builder/#dockerignore-file

2.3.2 N'installez rien de superflu

Afin de réduire la quantité de dépendances à installer, n'installez pas de paquets dont vous n'avez pas vraiment l'utilité : il n'y a pas de raison par exemple d'avoir un éditeur de texte dans un environnement qui sera utilisé comme serveur web. Un autre conteneur pourra contenir cet éditeur de texte dans les cas où vous avez besoin de modifier des données.

En plus, cela réduira le temps de construction et la taille des images produites !

Avec `apt` par exemple, vous pouvez ajouter l'option `--no-install-recommends` lors vous installer un paquet qui vient avec de nombreuses recommandations inutiles. C'est le cas par exemple de `ffmpeg` ou de `gstreamer`, qui viennent tous deux avec de nombreux *codecs*, mais peut-être que vous savez exactement de quels *codecs* vous avez besoin.

2.3.3 Minimisez le nombre de couches

Vous devez trouver l'équilibre idéal entre la lisibilité de votre `Dockerfile` (qui assure la maintenabilité sur le long terme) et le nombre de couches créées.

Utilisez les constructions en plusieurs étapes pour n'en recopier que les éléments utiles dans l'image finale. C'est le meilleur moyen de gagner de la place.

2.3.4 Ordonnez vos lignes de commandes complexes

2.3.4.1 Allez à la ligne pour séparer les longues lignes de commandes complexes

Aérez vos `Dockerfile` !

N'hésitez pas à commenter et séparer les blocs logiques ensemble, comme lorsque vous codez.

Lorsqu'une ligne devient complexe, allez à la ligne :

```
RUN apt-get update && apt-get install -y \
       nginx \
       php5-fpm
```

Notez les backslashs à la fin des lignes, indiquant qu'elle n'est pas terminée.

2.3.4.2 Triez les arguments par ordre alphabétique

Lorsque c'est possible, ordonnez vos lignes suivant un ordre logique. Par exemple :

```
RUN apt-get update && apt-get install -y \
        bzr \
        cvs \
        git \
        mercurial \
        subversion
```

2.3.5 Profitez du système de cache

Le processus de construction de votre image Docker va lire les informations de votre Dockerfile dans l'ordre. Pour chaque instruction, Docker va essayer de trouver si une image n'est pas déjà disponible dans le cache (plutôt que de créer une nouvelle image identique).

Il y a un certain nombre de règles à connaître pour bien utiliser ce mécanisme :

— En démarrant d'une image de base déjà présente dans le cache (`docker images`), l'instruction suivante est comparée avec toutes les autres images existantes qui en dérivent directement. Si aucune image correspondant n'est trouvée pour l'instruction, le cache est invalidé pour le reste de cette construction.
— Dans la plupart des cas, Docker va simplement comparer l'instruction lue avec le(s) différente(s) image(s) qui dérive(nt) de la commande précédente. Si aucune commande correspondante n'est trouvé, le cache se retrouve invalidé pour les instructions suivantes.
— Pour les instructions `ADD` et `COPY`, en plus de la comparaison précédente, la somme de contrôle du fichier est ajoutée. Si le fichier a été modifié, le cache se retrouve invalidé.
— Une fois que le cache est invalidé, toutes les commandes restantes à exécuter dans le `Dockerfile` vont être exécutées.

2.3.6 Concevez des conteneurs éphémères

Les conteneurs que vous générez doivent être aussi éphémères que possible : ils devraient pouvoir être arrêtés, détruits et recréés sans nécessiter d'étape de reconfiguration. La configuration devrait se faire au lancement du conteneur ou lors de sa construction.

2.3.7 Cas d'`apt-get` et des gestionnaires de paquets

— N'exécutez pas `apt-get update` seul sur une ligne. Cela risque de poser des problèmes de cache, car la ligne ne va jamais changer et le cache sera toujours utilisé. Vous risquez de récupérer des paquets qui ne sont pas à jour.
— Pour assurer une bonne gestion du cache, n'hésitez pas à indiquer les versions des programmes que vous voulez installer sur votre ligne de commande `apt-get`. Lors d'un changement de version, vous changerez la ligne, le cache ne sera donc pas utilisé.

2.3.8 Exposez les ports standards

La commande `EXPOSE` permet d'indiquer les ports sur lesquels votre conteneur s'attend à recevoir des paquets venant de l'extérieur. Ces ports ne sont pas partagés avec l'hôte ou les autres conteneurs, donc vous n'avez pas de raison de ne pas utiliser les ports standards.

Si vous faites cela, il y a de forte chance qu'il n'y ait pas besoin de modifier la configuration des autres logiciels contenu dans d'autres conteneurs puisqu'ils sont généralement configurés pour se connecter aux ports standards.

S'il y a un conflit sur la machine hôte, il sera toujours temps de créer une redirection à ce moment-là.

2.3.9 La bonne utilisation de l'`ENTRYPOINT`

L'*entrypoint* (on le verra plus en détail dans la partie suivante) peut être utilisé de deux manières différentes :

— Vous pouvez l'utiliser de telle sorte que la commande passée au `docker run`, après le nom de l'image, corresponde aux arguments attendu par le programme indiqué dans l'*entrypoint*. Par exemple pour `nginx` :

```
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon", "off;"]
```

— Vous pouvez aussi utiliser un script qui servira à faire les initialisations ou les configurations nécessaire au bon fonctionnement du conteneur (rappelez-vous, il doit être éphémère !). Par exemple, le `Dockerfile` pour l'image de PostgreSQL possède cet entrypoint :

```
#!/bin/bash
set -e

if [ "$1" = 'postgres' ]; then
chown -R postgres "$PGDATA"

if [ -z "$(ls -A "$PGDATA")" ]; then
  gosu postgres initdb
fi
```

```
exec gosu postgres "$@"
fi

exec "$@"
```

2.3.10 `[""]`, ' et sans `[]`

Les instructions `ENTRYPOINT` et `CMD` peuvent prendre deux formes :

— `["cmd", "arg1", "arg2"]` : ici, un simple `execve` sera effectué avec ces arguments. Si d'éventuels variables se trouve dans les arguments, elles ne seront pas remplacées.
— `cmd arg1 arg2` : ici l'exécution se fera au sein d'un `sh -c`, donc les variables seront remplacées et étendues.

Les commandes sous forme de tableau étant parsées par un parser JSON, vous ne pouvez pas utiliser les *simples quotes*.

2.3.11 Volumes

L'instruction `VOLUME` doit être utilisée pour exposer tous les espaces de stockage de données, configuration, ...

2.3.12 Réduisez les privilèges

Utilisez l'instruction `USER` dès que vous le pouvez, lorsqu'un service ne réclame pas de privilège particulier.

Il vous faudra sans doute créer l'utilisateur et son groupe dans le `Dockerfile`.

2.3.13 Profitez du système de liaison et de résolution de nom

Dès lors que vous effectuez un lien avec un autre conteneur, son nom (ou son alias) est ajouté au fichier `/etc/hosts`. Cela signifie que lorsqu'un nom de domaine correspondant au nom du conteneur (ou son alias) sera recherché, l'IP sera bien celle du conteneur. Lorsque vous configurez un conteneur, utilisez de préférence un nom plutôt qu'une IP, qui changera à coup sûr.

Au moment du `docker run`, vous pouvez préciser d'autres noms d'hôtes particuliers en utilisant l'option `--add-host`.

2.3.14 Exécutez un seul processus par conteneur

Dans la majorité des cas, vous ne devriez jamais lancer plus d'un seul processus par conteneur. Il est préférable de répartir chaque application dans un conteneur distinct qui n'effectue que le travail pour lequel il est chargé. Les options de liaison entre conteneur sont à votre disposition pour vous aider à cette tâche.

2.4 De l'intérêt de faire des images minimales

À l'inverse de langages comme Javascript, Python, Java et bien d'autres, le langage Go compile, comme le C, vers du code directement exécutable par le processeur. Tandis que les langages interprétés ont besoin de leur interpréteur et de leur environnement d'exécution, les langages compilés n'ont pas besoin d'être distribués avec leur compilateur.

Prenons le temps de regarder les tailles des images :

```
42sh$ docker image ls -f reference=golang -f reference=youp0m
REPOSITORY        TAG      IMAGE ID      CREATED      SIZE
golang            1-alpine 155ead2e66ca  3 months ago 328MB
nemunaire/youp0m  latest   2c06880e48aa  2 days ago   25MB
```

L'image contenant le compilateur Go est bien plus lourde que l'image minimale que l'on a construite avec le binaire compilé. C'est autant d'espace et de performances gagnées.

2.5 `buildx`

Docker `buildx` est un plugin qui apporte BuildKit. Tout en étant compatible avec la syntaxe des `Dockerfile` existant, BuildKit apporte une gestion concurrente des nœuds de construction : très utile lorsque l'on construit une image pour plusieurs architectures.

Installation Windows et MacOS

Avec Docker Desktop, le plugin est déjà installé, vous n'avez aucune action supplémentaire à effectuer, vous pouvez commencer à l'utiliser.

Installation Linux

En fonction de la méthode d'installation que vous avez suivie, vous avez peut-être déjà le plugin installé. Si vous n'avez pas d'erreur en exécutant `docker buildx`, mais que vous voyez l'aide de la commande, c'est bon. Sinon, vous pouvez l'installer comme ceci :

```
V="v0.9.1"
mkdir -p ~/.docker/cli-plugins
curl -L -s -S -o ~/.docker/cli-plugins/docker-buildx \
    https://github.com/docker/buildx/releases/download/$V/buildx-$V.linux-amd64
chmod +x ~/.docker/cli-plugins/docker-buildx
```

2.5.1 Utilisation

Nous pouvons réutiliser le `Dockerfile` que vous avez écrit pour `youp0m`, en remplaçant simplement la ligne de `docker build` par celle-ci :

```
docker buildx build .
```

(Nous ne rentrerons pas plus dans les détails de cette nouvelle commande, mais sachez qu'on la retrouve particulièrement fréquemment dans les *GitHub Actions* :)

(https://github.com/marketplace/actions/docker-setup-buildx)

2.5.2 `docker/dockerfile:1.4`

La version habituelle de la syntaxe des `Dockerfile` est la version 1.1. En utilisant BuildKit, nous pouvons dès à présent passer à la version 1.4.

Les ajouts par rapport à la syntaxe usuelle sont répertoriés sur cette page : https://hub.docker.com/r/docker/dockerfile.

Faites en sorte que le `Dockerfile` que vous avez créé pour `youp0m` indique un *frontend* BuildKit à utiliser, tout en restant compatible avec la syntaxe du `docker build` classique.

2.6 D'autres méthodes pour créer des images

Les images utilisées par Docker pour lancer les conteneurs répondent avant tout aux spécifications OCI. Le format étant standard, il est normal que d'autres outils puissent utiliser, mais aussi créer des images.

2.6.1 Changer la syntaxe de nos `Dockerfile`

Parfois on peut se sentir un peu frustré par la syntaxe des `Dockerfile` ou par son manque d'évolutivité. Avec BuildKit, il est possible de préciser un parseur à utiliser pour l'évaluation de la syntaxe du `Dockerfile`. Les parseurs (*frontend* dans la documentation anglaise) sont des images Docker : on indique leur nom dans un commentaire au tout début du fichier :

```
# syntax=docker/dockerfile:1.4
FROM ubuntu
RUN apt-get update && apt-get install gimp
```

La possibilité d'avoir plusieurs implémentations de `Dockerfile` apporte pas mal d'avantages :

— La version de l'image *frontend* est systématiquement comparée en ligne au début du parsing du `Dockerfile`, ce qui assure la récupération des derniers correctifs/versions, sans nécessiter une mise à jour du *daemon* Docker.
— On s'assure que chaque développeur/utilisateur utilise la même implémentation, avec la même version.
— L'évolution de la syntaxe peut être plus souple car elle ne dépend plus de la version de Docker installée, mais de la version déclarée dans le `Dockerfile`.
— On peut même créer de nouvelles syntaxes facilement.

Les images de parseur sont chargées, à partir d'un fichier lisible et compréhensible par un humain, de créer une représentation intermédiaire (*LLB*). Cette représentation intermédiaire se compose d'une liste d'opérations basiques (*ExecOp*, *CacheOp*, *SecretOp*, *SourceOp*, *CopyOp*, ...).

N'hésitez pas à jeter un œil aux autres langages de `Dockerfile` existants, notamment :

— Gockerfile : bien que dépassé faute de mise à jour, cette implémentation est très simple à comprendre : elle a pour but de créer une image contenant un unique binaire Go, à partir du nom de son dépôt.

— hlb : une syntaxe prometteuse, plus proche pour les développeurs.
— Earthly : qui tend à regrouper `Makefile`, `Dockerfile`, et autres scripts de CI et de tests.

2.6.2 Des images sans Docker

Il est aussi possible de se passer complètement de Docker. La plupart des outils qui sont capables de générer des images de machines virtuelles, sont aussi capables de générer des images Docker. Citons notamment :

— Buildah : https://github.com/containers/buildah/ (utilisé par `podman`),
— Buildpacks : https://buildpacks.io/,
— Hashicorp Packer : https://www.packer.io/docs/builders/docker,
— Nix et Guix : https://nix.dev/tutorials/building-and-running-docker-images,
— Kubler : https://github.com/edannenberg/kubler,
— et bien d'autres.

3 Le point d'entrée du conteneur

Le point d'entrée ou l'*entrypoint* correspond à la ligne de commande qui sera exécutée au lancement du conteneur. Deux paramètres de notre `Dockerfile` permettent d'agir sur cette ligne de commande : `CMD` et `ENTRYPOINT`.

— **`CMD`** est la commande par défaut : lorsqu'au moment de `run`, aucun paramètre n'est passé après le nom de l'image, le contenu du dernier `CMD` rencontré sera utilisé.
— **`ENTRYPOINT`**, s'il est défini, sera réellement exécuté, qu'il y ait ou non des arguments pour remplacer la ligne de commande. Lorsque des arguments sont passés ou qu'un `CMD`, ceux-ci sont passés en argument de l'`ENTRYPOINT`.

Par exemple, avec le `Dockerfile` suivant, construisant l'image `sample-echo` :

```
FROM ubuntu
CMD ["world"]
ENTRYPOINT ["/bin/echo", "Hello"]
```

Nous obtenons les résultats suivants :

```
42sh$ docker run sample-echo
Hello world
```

Dans ce premier cas, il n'y a pas d'argument après le nom de l'image, c'est donc le contenu de `CMD` qui est utilisé ; il est donc passé en argument à l'`ENTRYPOINT`. Concrètement, la première ligne de commande exécutée est :

```
["/bin/echo", "Hello", "world"]
```

Essayons maintenant avec des arguments :

```
42sh$ docker run sample-echo $USER
Hello neo
```

Le contenu de la variable `$USER`, interprété par notre shell, est utilisé à la place de `CMD`.

Si l'on a besoin d'exécuter un `ENTRYPOINT` différent, il reste la possibilité de le surcharger au moyen d'un argument :

```
42sh$ docker run --entrypoint /bin/sh sample-echo
01abc345# _
```

3.1 Personnalisation basique

Afin de faire bénéficier à nos utilisateurs d'une immersion parfaite, nous allons faire en sorte que notre image puisse être utilisée ainsi :

```
docker run -d -p 80:80 youp0m -bind :80
```

Plutôt que de laisser l'utilisateur se débrouiller avec le chemin interne dans lequel il va trouver le bon binaire :

```
docker run -d -p 80:80 youp0m /srv/youp0m -bind :80
```

Essayez les deux commandes, si vous avez utilisé l'instruction `CMD` dans votre `Dockerfile` jusqu'à présent, vous devriez vous trouver dans le deuxième cas.

Pour améliorer la situation, définissez l'`ENTRYPOINT` de votre image sur le binaire `/srv/youp0m`.

3.2 Point d'entrée avancé

Dans certains cas, il peut être nécessaire au lancement d'un conteneur de faire un minimum d'étapes d'initialisation avant que le conteneur ne soit opérationnel (rappelez-vous les options que l'on passait à l'image `mysql` pour créer un utilisateur et une base).

Notre but, dans cette partie, sera de créer un utilisateur administrateur (pouvant passer le contrôle d'accès http://localhost:8080/admin/) :

```
docker run -i --rm -p 8080:8080 -e YOUP0M_PASSWORD=admin youp0m
```

3.2.1 Bases du script

Notre script d'`ENTRYPOINT` sera appelé avec en argument, ceux passés par l'utilisateur après le nom de l'image, ou, à défaut, le contenu de `CMD`.

C'est donc l'`ENTRYPOINT` qui est responsable de la bonne utilisation de ceux-ci, de leur modification, …

À la fin d'un script d'`ENTRYPOINT`, afin de garder comme premier processus du conteneur le programme qui nous intéresse, on réalise un `execve(2)`, sans `fork(2)` :

```
exec /srv/youp0m $@
```

Dans cet exemple : `exec` est la commande interne à notre shell pour lui indiquer de remplacer son fil d'exécution par cette commande (sans `exec`, il va `fork(2)` avant). `$@` est ici pour transmettre tel quel la liste des arguments passés au script (il s'agit de ceux donnés par l'utilisateur, sur la ligne de commande du `run`, ou du contenu de `CMD` si l'utilisateur n'a rien précisé).

3.2.2 Format du fichier `htpasswd`

Le format attendu est celui d'un fichier `htpasswd` typique d'Apache. Nous pouvons obtenir un fichier valide avec :

```
(
    echo -n "$YOUP0M_USERNAME"
    echo -n ":"
    openssl passwd -crypt "$YOUP0M_PASSWORD"
) > myhtpasswd
```

Il faut ensuite passer le chemin du fichier créé sur la ligne de commande grâce à l'option `-htpasswd`.

Écrivez un script d'`ENTRYPOINT`, analysant les variables d'environnement, à la recherche de `YOUP0M_USERNAME` et `YOUP0M_PASSWORD` pour initialiser le fichier `.htpasswd` qui sera ajouté à la liste des arguments à passer au service.

Par exemple :

```
42sh$ docker run -d -p 8081:8081 -e YOUP0M_USERNAME=admin \
                -e YOUP0M_PASSWORD=admin youp0m -bind=:8081

42sh$ curl -u admin:badpasswd http://localhost:8081/admin/
You are not allowed to perform this request.

42sh$ curl -u admin:admin http://localhost:8081/admin/
<!DOCTYPE html>
```

3.3 Étendre un `ENTRYPOINT` existant

Vous venez de réaliser un script d'*entrypoint* pour votre conteneur. Il ajoute assurément de nombreuses fonctionnalités indispensables. Mais que se passe-t-il si quelqu'un souhaite étendre votre image, ou simplement pour ajouter une fonctionnalité ?

La plupart des images officielles[3] prêtes à l'emploi disposent d'un dossier `/docker-entrypoint.d`, à la racine de l'image ; et d'un script d'*entrypoint* qui va se charger d'appeler chacun des scripts du dossier avant de lancer la commande par défaut.

Chaque fonctionnalité distincte de l'*entrypoint* est placée dans un script séparé, et quelqu'un qui souhaite ajouter son propre script peut le faire facilement, soit au moyen d'un volume, soit en étendant l'image.

[3]Consultez le dépôt de l'image `nginx` par exemple. Il possède 3 scripts pour 3 fonctionnalités différentes.

4 Analyse de vulnérabilité

Nous avons vu jusqu'à présent que Docker nous apportait un certain degré de sécurité d'emblée, au lancement de nos conteneurs. Cela peut sans doute paraître quelque peu rassurant pour une personne chargée d'administrer une machine hébergeant des conteneurs, car cela lui apporte des garanties quant à l'effort de cloisonnement mis en place.

Mais doit-on pour autant s'arrêter là et considérer que nous avons réglé l'ensemble des problématiques de sécurité liées aux conteneurs ?

Évidemment, non : une fois nos services lancés dans des conteneurs, ils ne sont pas moins exposés aux bugs et autres failles applicatives ; qu'elles soient dans notre code ou celui d'une bibliothèque, accessible par rebond, ...

Il est donc primordial de ne pas laisser ses conteneurs à l'abandon une fois leur image créée et envoyée en production. Nos conteneurs doivent être regénérés sitôt que leur image de base est mise à jour (une mise à jour d'une image telle que Debian, Ubuntu ou Redhat n'apparaît que pour cela) ou bien lorsqu'un des programmes ou l'une des bibliothèques que l'on a installées est mise à jour.

Convaincu ? Cela sonne encore comme des bonnes pratiques difficiles à mettre en œuvre, pouvant mettre en péril tout un système d'information. Pour s'en protéger, nous allons avoir besoin de réaliser à intervalles réguliers une analyse statique de nos conteneurs.

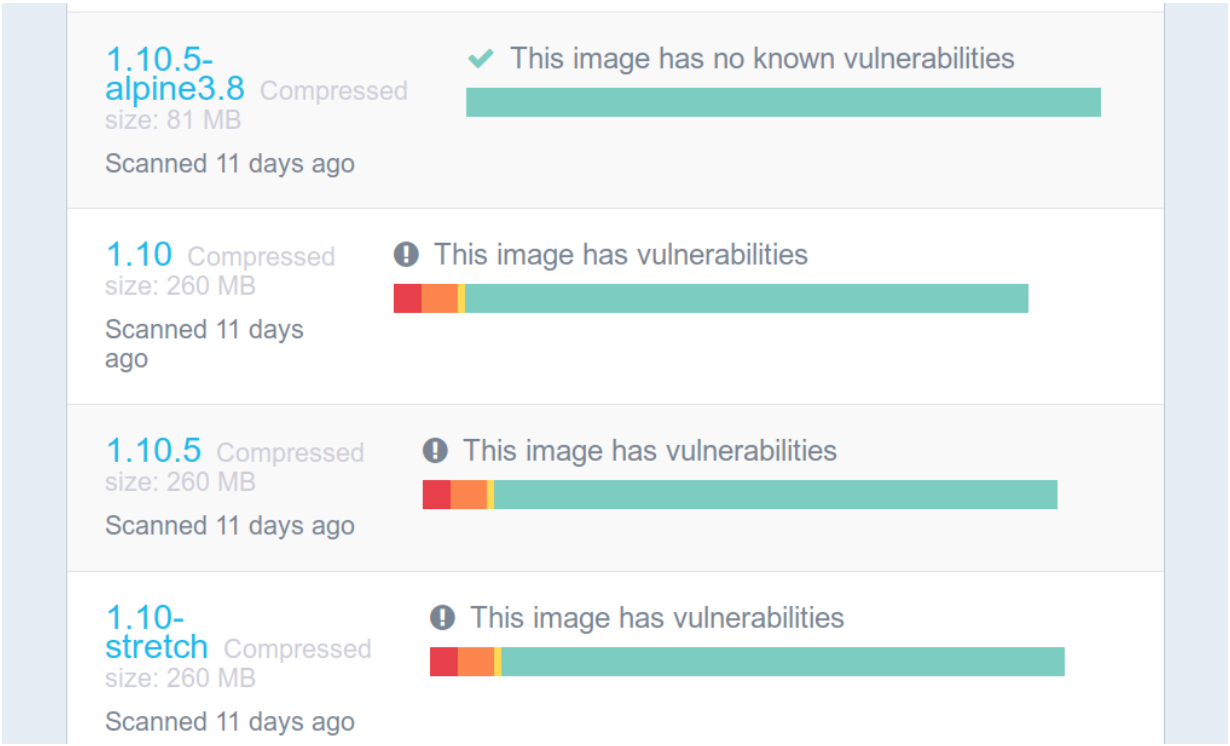

Figure 3: Scan de vulnérabilités sur le Docker Hub

Selon une étude[4] réalisée sur les images du Docker Hub, elles présenteraient en moyenne 180 vulnérabilités, beaucoup ne sont pas mises à jour depuis trop longtemps et les vulnérabilités ont surtout tendance à se propager à cause de l'usage d'une image parente pas à jour.

[4] https://www.enck.org/pubs/shu-codaspy17.pdf

Une mesure efficace consiste à reconstruire régulièrement (et surtout automatiquement) les images que l'on publie sur un registre, sans oublier de mettre à jour l'image de base.

D'ailleurs, avez-vous vérifié qu'une mise à jour de l'image `registry.nemunai.re/youp0m` n'était pas disponible depuis que vous avez commencé à l'utiliser ? Docker ne vérifie jamais si une mise à jour des images que vous avez précédemment téléchargées. Pensez donc régulièrement à appeler :

```
42sh$ docker image pull IMAGE
```

4.1 Docker Scan

Pour faire face à ces problèmes de sécurité qui prennent de l'ampleur, le Docker Hub, dans son modèle payant, permet d'analyser régulièrement ses images, pour avoir une idée sur la nécessité de les reconstruire.

Un plugin existe pour réaliser des scans d'images présentes sur votre machine, à travers les données du Docker Hub. Cela nécessite d'avoir un compte Docker, si vous n'en avez pas, nous verrons dans la section suivante `trivy` qui permet de réaliser ses scans directement sur notre machine, sans passer par un intermédiaire.

Par cette méthode, vous êtes limité à 10 scans par mois avec un compte gratuit.

4.1.1 Installation du plugin

Windows et MacOS Avec Docker Desktop, le plugin est déjà installé. Il faut que vous vous soyez préalablement connecté à votre compte Docker avec la commande `docker login`.

Linux Comme `docker scan` est un plugin, suivant la méthode d'installation que vous avez suivie, il n'a pas forcément été installé. Si vous obtenez un message d'erreur en lançant la commande, voici comment récupérer le plugin et l'installer manuellement :

```
mkdir -p ~/.docker/cli-plugins
curl -L -s -S -o ~/.docker/cli-plugins/docker-scan \
    https://github.com/docker/scan-cli-plugin/releases/\
    latest/download/docker-scan_linux_amd64
chmod +x ~/.docker/cli-plugins/docker-scan
```

4.1.2 Utilisation

Une fois le plugin installé et la licence du service acceptée, nous pouvons commencer notre analyse :

```
42sh$ docker scan nemunaire/youp0m

Testing nemunaire/youp0m...

Package manager:   apk
Project name:      docker-image|nemunaire/youp0m
Docker image:      nemunaire/youp0m
Platform:          linux/amd64
```

```
Base image:        alpine:3.16.2

 Tested 15 dependencies for known vulnerabilities, no vulnerable paths found.

According to our scan, you are currently using the most secure version of
the selected base image
```

```
$ docker scan mysql

Testing mysql...

 Low severity vulnerability found in util-linux/libuuid1
[...]

 High severity vulnerability found in gcc-8/libstdc++6
  Description: Insufficient Entropy
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-GCC8-469413
  Introduced through: apt@1.8.2.3, mysql-community/mysql-client@[...]
  From: apt@1.8.2.3 > gcc-8/libstdc++6@8.3.0-6
  From: mysql-community/mysql-client@8.0.26-1debian10 > gcc-8[...]
  From: mysql-community/mysql-server-core@8.0.26-1debian10 > gcc-8[...]
  and 7 more...
  Image layer: Introduced by your base image (mysql:8.0.26)

Package manager:   deb
Project name:      docker-image|mysql
Docker image:      mysql
Platform:          linux/amd64
Base image:        mysql:8.0.30

Tested 119 dependencies for known vulnerabilities, found 24 vulnerabilities

According to our scan, you are currently using the most secure version of
the selected base image
```

Ce dernier exemple est sans appel : `mysql` est une image officielle, et sa dernière version à l'écriture de ses lignes contient pas moins de 24 vulnérabilités dont 9 *high* (pourtant corrigées dans des versions suivantes).

4.2 Trivy

Le principal outil pour chercher des vulnérabilités connues dans les images Docker est `trivy`, édité par Aqua security.

À partir des informations mises à disposition par les équipes de sécurité des principales distributions, `trivy` va générer un rapport, pour chacune de nos images, indiquant les vulnérabilités connues au sein de l'image.

L'outil se présente sous la forme d'un binaire ou d'une image Docker, prenant un certain nombre d'arguments, notamment le nom de l'image à analyser.

4.2.1 Utilisation

Tentons à nouveau d'analyser l'image `mysql` :

```
42sh$ docker run --rm aquasec/trivy image mysql
INFO Need to update DB
INFO Downloading DB... 100.00% 14.41 MiB p/s 2s
INFO Detected OS: oracle
INFO Detecting Oracle vulnerabilities...
INFO Number of language-specific files: 0

mysql (oracle 8.6)
==================
Total: 5 (UNKNOWN: 0, LOW: 0, MEDIUM: 4, HIGH: 1, CRITICAL: 0)
```

Les résultats sont un peu différents qu'avec `docker scan`, mais on constate que l'image `mysql` contient vraiment de nombreuses vulnérabilités. Même si elles ne sont heureusement pas forcément exploitables directement.

Voyons maintenant s'il y a des différentes avec l'image `nemunaire/youp0m` :

```
42sh$ docker run --rm aquasec/trivy image nemunaire/youp0m
INFO Need to update DB
INFO Downloading DB... 100.00% 15.98 MiB p/s 1s
INFO Detected OS: alpine
INFO Detecting Alpine vulnerabilities...
INFO Number of language-specific files: 1
INFO Detecting gobinary vulnerabilities...

nemunaire/youp0m (alpine 3.16.2)
================================
Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)


srv/youp0m (gobinary)
=====================
Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)
```

Nous pouvons remarque que Trivy, en plus de faire l'analyse statique des vulnérabilités de l'image, a aussi fait une analyse des dépendances du binaire `/srv/youp0m`.

Trivy est en effet capable de rechercher des vulnérabilités par rapport aux dépendances connues de certains langages : Python, PHP, Node.js, .NET, Java, Go, …

4.2.2 Usage du cache

Pour éviter de surcharger les serveurs de distributions de la base de données de vulnérabilités, nous devrions utiliser un cache pour faire nos analyses. Préférez lancer `trivy` avec les options suivantes :

```
42sh$ docker run --rm -v /tmp/trivy-cache:/root/.cache/ aquasec/trivy \
         image IMAGE
```

4.2.3 Format de génération du rapport

Lorsque nous appelons `trivy` directement, il génère un rapport au format texte lisible directement dans notre terminal. Il peut être pourtant pratique de pouvoir l'exporter pour l'afficher dans un navigateur (par exemple pour le mettre à disposition des développeurs, lors d'une analyse automatique).

Pour ce faire, on peut ajouter les options suivantes à la ligne de commande de notre conteneur :

```
--quiet --format template --template "@contrib/html.tpl"
```

En redirigeant la sortie standard vers un fichier, vous pourrez l'ouvrir dans votre navigateur favori.

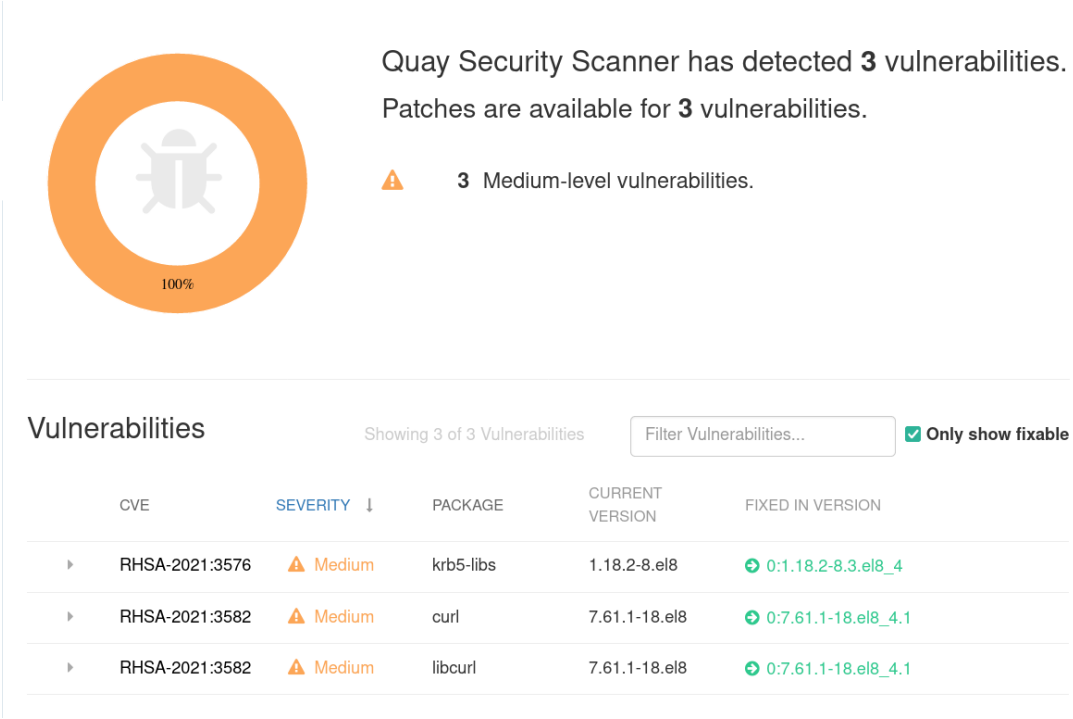

Figure 4: Scan de vulnérabilités sur le registre Quay.io

4.3 Clair

`Clair` est un projet de CoreOS, similaire à Trivy : il va rechercher parmi sa base de vulnérabilités lesquels concernent les images qu'on lui donne.

Contrairement à Trivy, il ne va pas tenter d'analyser les dépendances supplémentaires des applications métiers (Trivy est capable d'analyser les dépendances PHP, Python, Ruby, Go, ...).

La mise en œuvre de Clair est un peu plus complexe car il s'agit d'un *daemon* qui s'exécute en permanence, et auquel on peut transmettre, via une API, nos images. Dans un contexte d'intégration continue, ou d'un registre d'images qui teste régulièrement ses images hébergées, c'est un outil idéal. Néanmoins il faut le configurer un minimum pour qu'il soit opérationnel. Consultez l'annexe dédiée si vous souhaitez opter pour cette solution.

On notera tout de même que les outils donnés génèrent des rapports HTML avec des graphiques explicites :

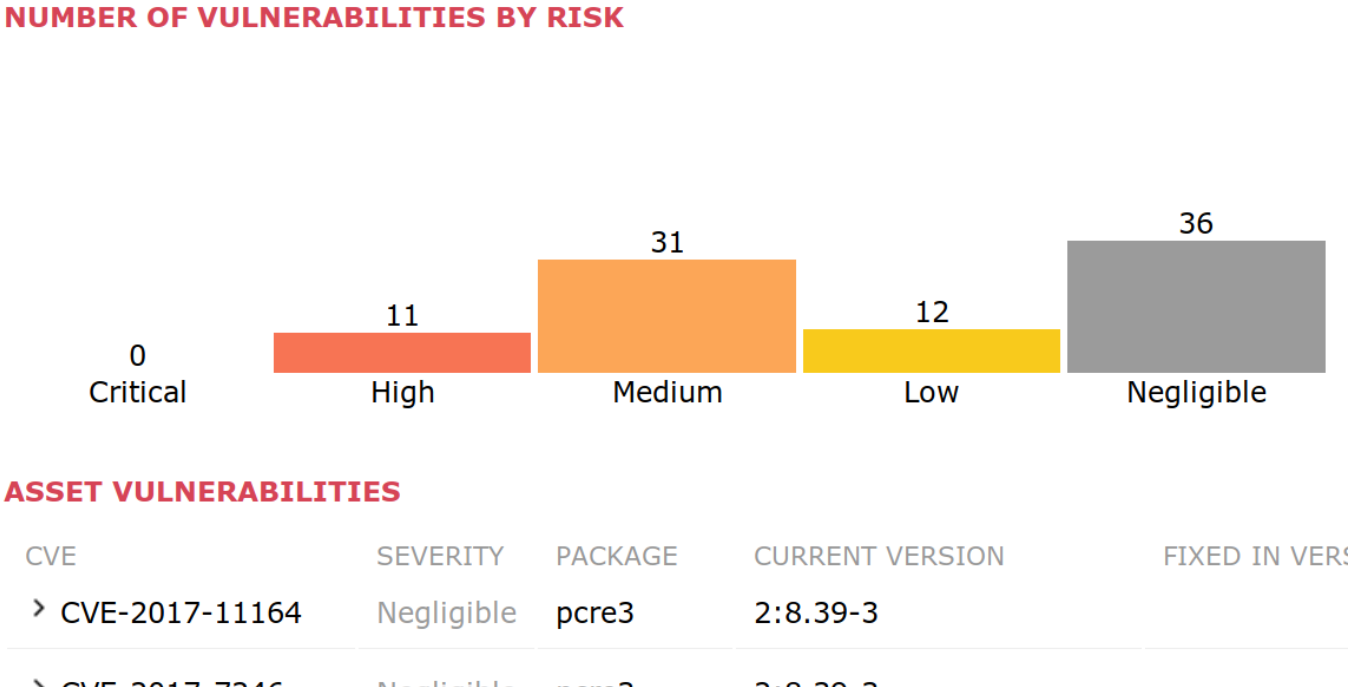

Figure 5: Rapport d'analyse statique des vulnérabilités par Clair

Déterminez le nombre de vulnérabilités dans les principales images officielles du Docker Hub.

En utilisant l'outil de votre choix, exporter un rapport HTML (ou la sortie standard de l'outil s'il n'exporte pas en HTML) de l'image **officielle** contenant les vulnérabilités les plus inquiétantes. Vous placerez ce rapport dans un fichier `${IMAGE}:${TAG}.html` ou `${IMAGE}:${TAG}.txt`.

Veillez à utiliser une image **différente** de `mysql`, utilisée en exemple.