Audit de sécurité d'un environnement Docker

Julien Raeis et Matthieu Buffet
julien.raeis@ssi.gouv.fr
matthieu.buffet@ssi.gouv.fr

Agence nationale de la sécurité des systèmes d'information

1 Introduction

D'après le Trésor de la Langue Française informatisé[1] :

CONTAINER substantif masculin.
TRANSPORT, emploi courant. Caisse de forte capacité et de dimensions normalisées, destinée à faciliter les opérations de manutention, notamment en évitant les ruptures de charge d'un mode de transport à un autre.
Rem. 1. La forme conteneur a été proposée pour remplacer cet anglicisme.

L'isolation des environnements d'exécution de processus, pour des raisons de portabilité ou de sécurité, est une problématique courante, en particulier depuis l'avènement des hébergements externalisés d'applications dans ce qu'on appelle aujourd'hui communément le *cloud*.

Afin de répondre à cette problématique, les systèmes d'exploitation et certains éditeurs proposent des solutions de virtualisation dont les propriétés doivent aujourd'hui inclure, au-delà de la sécurité, la facilité de déploiement et le passage à l'échelle. Parmi ces solutions on retrouve, outre les hyperviseurs classiques tels Microsoft Hyper-V ou KVM, des technologies dites de « virtualisation légère », basées sur des mécanismes d'isolation de processus propres à chaque système d'exploitation.

De nombreux produits du marché, commerciaux ou non, exploitent ces capacités et sont aujourd'hui très largement utilisés. Des logiciels comme Docker, LXC ou encore Kubernetes font le quotidien de nombreux professionnels de l'informatique.

La sécurité de ces systèmes est fortement intriquée avec celle du système d'exploitation les hébergeant. Cependant, bien que cette sécurité ait fait l'objet de nombreuses publications [1–3, 9], les méthodologies permettant de l'évaluer n'ont été que très superficiellement étudiées. Il convient d'abord de faire un rappel des technologies mises en œuvre, à la fois sous Linux mais également sous Windows, puis de passer en revue divers points liés à la sécurité de leur configuration ou de leur déploiement.

[1] http://www.atilf.fr/tlfi

2 Définitions

Afin de disposer d'un vocabulaire commun, voici quelques définitions de concepts tels qu'ils seront utilisés dans le présent article.

Conteneur : processus ou groupe de processus (généralement un parent et ses enfants), exécutés dans un contexte restreint en matière de visibilité et d'accès aux ressources du système d'exploitation. Plusieurs conteneurs peuvent être exécutés par un même noyau.

Système hôte : système d'exploitation exécutant des conteneurs, n'étant pas lui-même exécuté dans un conteneur. Il peut cependant être exécuté dans une machine virtuelle.

Hyperviseur : système d'exploitation exécutant des machines virtuelles. Ceci inclut VMware ESXi, Microsoft Hyper-V, Xen ou encore KVM.

Gestionnaire de conteneurs : logiciel instanciant des conteneurs et mettant en œuvre des mécanismes d'isolation et de sécurité au niveau du système hôte. Docker ou LXD en sont des exemples.

Orchestrateur : logiciel de déploiement et de gestion du cycle de vie des conteneurs au travers des gestionnaires de conteneurs. On retrouve Docker Swarm ou encore Kubernetes dans cette catégorie.

Image : système de fichiers statique, créé souvent automatiquement, représentant un conteneur avant instanciation. Une image peut être constituée de couches successives en lecture seule ou n'être qu'une archive contenant des fichiers.

Registre : dépôt d'images, souvent accessible au moyen du protocole HTTP, utilisé par le gestionnaire de conteneurs comme source pour instancier des conteneurs.

3 Historique

L'idée de vouloir isoler des processus dans des environnements d'exécution restreints n'est pas neuve. L'appel système `chroot()` a été introduit en 1979 dans UNIX pour répondre à cette problématique, en hébergeant plusieurs versions d'un même programme, chacune disposant d'une vue propre de la racine du système de fichiers. Les objectifs étaient de faciliter le développement au travers d'une meilleure gestion des dépendances et d'améliorer la compatibilité avec des versions antérieures de bibliothèques. Ce concept est aujourd'hui encore l'une des bases des systèmes modernes de conteneurs.

En 2000, FreeBSD 4.0 intègre `jail` qui va au-delà de la simple restriction au système de fichiers. Ainsi `jail` exécute un processus avec :

– son propre nom de machine ;
– sa propre adresse IP et sa table de routage ;
– une sous-arborescence dédiée du système de fichiers, comme `chroot()` ;
– sa base d'utilisateurs et notamment un superutilisateur dédié.

En 2005, Sun Microsystems introduit les *Solaris Containers* avec Solaris 10 et en particulier son mécanisme d'isolation *Solaris Zones*. On y retrouve des fonctionnalités d'isolation comparables à celles de `jail`, mais les *Solaris Zones* permettent également de restreindre les ressources du système (mémoire, temps processeur, etc.) auxquelles ont accès chaque processus ou groupe de processus.

Enfin en 2008, faisant suite aux idées introduites par *Linux VServer* (2001) et *OpenVZ* (2005), le noyau Linux 2.6.24 intègre la fonctionnalité *cgroups*, ouvrant la voie à ce qui est aujourd'hui appelé *Linux Containers* (LXC). Les *cgroups*, en collaboration avec les *namespaces*, apparus à la même époque, sont aujourd'hui les briques de base de la plupart des systèmes de conteneurs couramment rencontrés sous Linux.

De leur côté, Apple et Microsoft ne sont pas en reste. En effet, Apple fournit depuis MacOS X 10.7 (2011) un mécanisme appelé *app sandboxing*[2]. Microsoft a entrepris de nombreux efforts de durcissement depuis la sortie de Windows Vista (2007) pour restreindre les capacités d'exécution des codes malveillants, par exemple via l'ajout des *Protected Processes* (généralisés en *Protected Processes Light* ou PPL sous Windows 8.1) : des processus protégés par le noyau et signés avec des autorités de certification spécifiques[3]. Le but ici est de les protéger des autres processus : il est par exemple impossible d'injecter un *thread* ou d'accéder à la mémoire virtuelle d'un PPL, même depuis un processus tournant sous l'identité `LocalSystem` et disposant du privilège `SeDebugPrivilege`. Seul le noyau ou un autre PPL pourra contourner ces mesures.

Plus récemment, les *AppContainers* ajoutés à Windows 8 rentrent, eux, dans la catégorie des fonctionnalités de « bac à sable ». À l'origine dévolus aux applications *Metro* et à Internet Explorer 10 et 11, leur utilisation est aujourd'hui possible pour l'ensemble des applications Windows. Chaque application demande l'accès à des ressources (connexion à Internet, ouverture d'un port en écoute, accès aux bibliothèques de photos, à la webcam, etc.) sous la forme d'une liste de *capabilities*[4], à ne pas confondre avec

[2] `https://developer.apple.com/app-sandboxing/`
[3] `http://download.microsoft.com/download/a/f/7/af7777e5-7dcd-4800-8a0a-b18336565f5b/process_vista.doc`
[4] `https://msdn.microsoft.com/en-us/library/windows/desktop/hh448474(v=vs.85).aspx`

leurs homonymes sous Linux, ressemblant aux permissions Android. Par défaut l'application a alors un jeton de sécurité *LowBox* combinant un SID unique à l'application, le SID de l'utilisateur, un SID par *capability*, et un niveau d'intégrité bas qui ne lui donne un accès en écriture qu'à ses ressources propres. L'objectif ici est de restreindre les capacités de latéralisation d'un code exploitant une vulnérabilité dans une telle application et toute communication ou tentative d'accès avec l'extérieur devra passer par un intermédiaire (*broker*) qui dispose de droits étendus, comme c'est le cas dans le bac à sable d'Internet Explorer.

Avec Windows Server 2016, Microsoft propose les *Windows Containers*[5], développés en collaboration avec Docker Inc. et décrits dans la suite de cet article.

4 Modélisation des menaces

Afin de définir quelles menaces peuvent peser sur des conteneurs, intéressons-nous d'abord à leurs cas d'usage les plus fréquents.

4.1 Cas d'usage fréquents

Trois cas d'usage sont fréquemment rencontrés dans l'emploi des technologies de conteneurs :

1. Packaging d'applications : permet de s'affranchir de l'installation de dépendances particulières sur le système hôte. Le programme (et potentiellemement son environnement) ainsi packagé ne l'est pas pour des raisons de sécurité, mais pour des raisons de facilité de déploiement, en abstrayant le système sous-jacent. Ces applications sont souvent exécutées avec les mêmes droits que si elles l'avaient été sur l'hôte.
2. Environnement de développement/compilation : simplifie la gestion des dépendances et la reproductibilité des compilations. Le code qui est exécuté est souvent maîtrisé et le but ici n'est pas de se protéger contre la malveillance.
3. Mécanisme d'isolation de services : restreint le contexte d'exécution de services, souvent accessibles par le réseau et soumis à des entrées d'utilisateurs non maîtrisées. C'est le cas typique des bacs à sable ou de l'hébergement mutualisé.

De ces cas d'usage, il est possible de déduire plusieurs modèles de menaces en fonction de ce qu'un utilisateur malveillant souhaiterait réaliser.

[5] `https://docs.microsoft.com/en-us/virtualization/windowscontainers/`

4.2 Modèles de menaces

Dans les deux premiers cas d'usage évoqués ci-dessus, la menace principale est probablement la perte d'intégrité des applications ou des conteneurs au travers par exemple de la falsification d'une image. Celle-ci pourrait intervenir dans les images sur lesquelles est basé le développement, ou entre les étapes de développement et de packaging réalisées par le développeur, ou encore au moment de la distribution des images. Une image modifiée pourrait contenir des composants malveillants, rajoutés dans le but de piéger l'application. Cette menace s'est concrétisée récemment avec l'exemple de plusieurs images du Docker Hub contenant des portes dérobées et mineurs de cryptomonnaie, téléchargées plus de 5 millions de fois pendant un an malgré plusieurs signalements[6].

Dans le troisième cas, on considère un attaquant extérieur, n'ayant pu interagir avec l'image du conteneur entre le moment de sa création et son déploiement effectif. La menace principale est alors la perte d'isolation, soit du conteneur vers l'hôte, soit entre deux conteneurs.

Enfin, il convient de considérer une dernière menace, commune à tous les cas d'usage : la prise de contrôle de l'écosystème, par compromission du système hôte ou de l'un des composants de l'environnement (orchestrateur ou gestionnaire de conteneurs). Un attaquant pourrait alors altérer la disponibilité, l'intégrité ou la confidentialité des conteneurs, mais également des données qui y sont traitées.

5 Conteneurs sous Linux

Les conteneurs emploient des mécanismes d'isolation des processus implémentés par le noyau du système hôte. Sous Linux, ceux-ci ont pour objectif soit de contrôler les ressources que les processus peuvent utiliser (*cgroups*), soit de restreindre la vue qu'ils ont du reste du système (*namespaces*).

S'ajoutent à cela des mesures de sécurité telles que *seccomp*, les *capabilities* ou l'application d'une politique de sécurité au travers de systèmes de type *Mandatory Access Control* (MAC) comme *AppArmor* ou *SELinux*. L'objectif de ces mesures est de limiter les conséquences d'une exploitation potentielle de vulnérabilité dans le contexte d'un conteneur, en particulier vis-à-vis de l'étanchéité des environnements.

Les mécanismes d'isolation ou de sécurité implémentés sous Linux ont déjà été très largement étudiés dans la littérature. En conséquence, ils ne seront décrits que sommairement ici.

[6] `https://github.com/docker/hub-feedback/issues/1121`

5.1 Mécanismes d'isolation

***Control groups* (*cgroups*)** En 2006, deux ingénieurs de Google implémentent un système de gestion des ressources par processus ou groupe de processus, appelé à l'époque *process containers*. Ce système sera renommé *control groups* (ou *cgroups*) un an après et intégré au noyau Linux 2.6.24.

Un *cgroup* est un ensemble de processus qui sont liés entre eux par un critère arbitraire (par exemple une *slice* `systemd`) et auxquels sont appliqués un jeu de paramètres ou de limites concernant les ressources accessibles du système d'exploitation. Ces groupes sont la plupart du temps hiérarchiques et héritent de leur parent.

La manipulation des *cgroups* est faite par l'intermédiaire de contrôleurs, au travers d'une interface basée sur le système de fichiers virtuel et généralement disponible sous le point de montage `/sys/fs/cgroup/`.

Les *cgroups* permettent notamment de gérer les ressources suivantes :

- l'allocation de parts CPU (`cpuset`, `cpu`, `cpuacct`) ;
- l'allocation de mémoire (`memcg`) ;
- l'accès aux périphériques (`devcg`) ;
- les ressources réseau (`net_cls`, `net_prio`).

En particulier, le contrôleur `devcg` peut restreindre l'accès à certains périphériques au travers de listes blanches et/ou de listes noires.

Il est également possible de limiter les effets d'un déni de service au moyen des contrôleurs agissant sur la mémoire ou les processus, en imposant des quotas d'utilisation.

Les *cgroups* permettent de gérer les ressources telles que l'allocation de parts CPU ou de mémoire, l'accès à certains périphériques ou encore les ressources réseau. Ces paramétrages peuvent limiter les effets d'un déni de service, en imposant des quotas d'utilisation.

Namespaces Les *namespaces* (ou espaces de nommage) sont des mécanismes d'isolation des processus restreignant leur vue respective du système hôte. Trois appels système peuvent être utilisés pour interagir avec les *namespaces* :

- `clone()` : créé un nouveau processus et l'attache à un ou plusieurs *namespaces* ;
- `unshare()` : créé un *namespace* et y attache un processus existant [7] ;
- `setns()` : attache un processus à un *namespace* déjà existant.

Dans le noyau 4.14, il existe sept *namespaces* différents, représentés par des drapeaux qu'il est possible de passer aux appels systèmes ci-dessus. Ils sont décrits dans le tableau 1.

Pour implémenter leur isolation vis-à-vis du système hôte, les conteneurs utilisent fortement les *namespaces* de type *mount*, *pid*, *net* et *user*. Ce dernier a par ailleurs fait couler beaucoup d'encre quant à sa plus value suite à la découverte de plusieurs vulnérabilités découlant de la complexité de son implémentation [8].

Nom	**Périmètre**	**Drapeau de `clone()`**	**Noyau**	**Capacité demandée**
mount	points de montage	`CLONE_NEWNS`	2.4.19	`CAP_SYS_ADMIN`
uts	nom de machine	`CLONE_NEWUTS`	2.6.19	`CAP_SYS_ADMIN`
ipc	POSIX message queues, SystemV IPC	`CLONE_NEWIPC`	2.6.19	`CAP_SYS_ADMIN`
pid	PID	`CLONE_NEWPID`	2.6.24	`CAP_SYS_ADMIN`
net	réseau	`CLONE_NEWNET`	2.6.29	`CAP_SYS_ADMIN`
user	UID/GID	`CLONE_NEWUSER`	3.8	aucune
cgroup	*cgroups*	`CLONE_NEWCGROUP`	4.6	`CAP_SYS_ADMIN`

Tableau 1. Liste des *namespaces* dans le noyau Linux 4.14

Historiquement, Eric Biederman, développeur noyau ayant contribué aux *namespaces*, avait évoqué dès 2006 l'implémentation de *namespaces* dédiés à la sécurité lors de la conférence OLS 2006 [10]. Cette idée fut par la suite abandonnée au profit des *user namespaces*.

5.2 Mécanismes de sécurité

Seccomp Contrairement aux machines virtuelles qui doivent passer par des *hypercalls* exposés par leur hyperviseur, les conteneurs ont un accès direct aux appels systèmes exposés par le noyau du système hôte. Il serait ainsi possible à un attaquant d'exploiter toute vulnérabilité présente au travers de l'un de ces *syscalls* afin d'élever ses privilèges, voire de sortir du conteneur. *Seccomp*, apparu avec le noyau 2.6.12 en 2005, offre la possibilité de restreindre les appels systèmes que peuvent utiliser les processus et limite par la même occasion l'exploitabilité des failles les mettant en jeu.

L'activation de *seccomp* est réalisée au travers des appels système `seccomp()`, ou `prctl()` avec le paramètre `PR_SET_SECCOMP`. Par défaut, la première version de *seccomp* n'autorisait qu'une liste blanche fixe contenant les appels systèmes `_exit()`, `read()`, `write()` et `sigreturn()`.

La seconde version, appelée **seccomp-bpf** et apparue en 2012 avec Linux 3.5, autorise la création de filtres *Berkeley Packet Filter* (BPF) afin de décrire des politiques de filtrage d'appels système plus fines. Des

filtres par défaut sont appliqués par les gestionnaires de conteneurs comme Docker ou LXD.

Capabilities L'objectif des *capabilities* (ou « capacités ») est de subdiviser les pouvoirs du superutilisateur `root`, notamment pour restreindre les conséquences de la compromission d'un programme privilégié. Un processus hérite généralement des *capabilities* de son parent. Dans le contexte des conteneurs, ce parent étant la plupart du temps un processus exécuté en tant que `root`, il en résulte que le processus possédant le PID 1 à l'intérieur du conteneur héritera de l'ensemble des *capabilities* disponibles. Les *capabilities* sont toujours détenues relativement à un *user namespace* (par défaut celui du PID 1 de l'hôte, quand ce namespace n'a pas été configuré), mais elles peuvent dans tous les cas donner de nombreux droits sur l'ensemble de l'hôte.

Les gestionnaires de conteneurs emploient généralement une liste blanche de *capabilities* pour n'en autoriser que le strict minimum nécessaire à l'exécution correcte du conteneur.

***Mandatory Access Control* (MAC)** Dans le cas où un conteneur serait compromis et où un attaquant arriverait à briser l'isolation vis-à-vis de l'hôte, le noyau dispose encore de mécanismes afin de contraindre voire d'empêcher la propagation de l'attaque. Ces systèmes de MAC permettent la création de profils destinés aux gestionnaires de conteneurs sous forme de liste blanche d'autorisations d'accès à des ressources du système, telles que le système de fichiers ou les interfaces réseau.

Parmi eux, *AppArmor* et *SELinux* sont les plus répandus. Des profils de sécurité pour Docker sont par exemple appliqués par défaut via SELinux sous CentOS et Fedora, et via AppArmor sous Debian et Ubuntu.

6 Conteneurs sous Windows

Disposant déjà d'un hyperviseur intégré à la version serveur de Windows depuis 2008, Microsoft a réalisé un partenariat avec la société Docker Inc. en 2014 afin de proposer une interface de gestion de conteneurs qui se rapproche de ce qui existait déjà sous Linux.

À partir de Windows Server 2016 (version 1607), trois types de conteneurs peuvent être instanciés en utilisant Docker sous Windows, sans exécuter le service Docker dans une machine virtuelle (la solution *Docker Toolbox*, maintenant dépréciée) :

- des conteneurs Linux, au travers d'une machine virtuelle Hyper-V (*MobyLinuxVM*) ;
- des conteneurs Windows (*Windows Server Containers* ou WSC), par des mécanismes d'isolation bas niveau du noyau de Windows Server proches de ceux présents sous Linux ;
- des conteneurs Windows (*Hyper-V Containers*), au travers d'une machine virtuelle minimale Hyper-V (*Utility VM*) exécutant un noyau Windows Server qui utilise les mêmes mécanismes que les WSC, mais ajoute une couche d'isolation par virtualisation.

Le démon `dockerd` et ses clients ont été portés sous Windows sans changement d'architecture, l'utilisation du langage Go aidant à la compatibilité multi plates-formes. Cependant les composants `docker-containerd` et `runc` étant spécifiques à l'environnement Linux, c'est le *Host Compute Service* (HCS) installé par le rôle Hyper-V qui s'occupe d'instancier les conteneurs et éventuelles machines virtuelles associées :

```
PS C:\> Get-Service vmcompute | fl -Property DisplayName,Status
DisplayName : Hyper-V Host Compute Service
Status      : Running
```

On notera une différence majeure avec les conteneurs Linux : l'arborescence des processus systèmes nécessaires à Windows (`wininit.exe`, `csrss.exe`, plusieurs `svchost.exe`, `lsass.exe`, etc.) est réinstanciée dans chaque conteneur. On verra que cela aura des conséquences sur le durcissement : par exemple, cela rend difficile le filtrage d'appels systèmes à l'échelle d'un conteneur entier comme le fait *seccomp* sous Linux.

6.1 Modes d'isolation

Là où tous les conteneurs Linux courants partagent le même noyau sur un même hôte, Windows fournit deux modes d'isolation différents, qui peuvent être utilisés simultanément.

Mode process Le mode *process*, utilisé par les *Windows Server Containers*, instancie un conteneur directement sur l'hôte et utilise des fonctions d'isolation fournies par le noyau de ce dernier, aussi bien pour le contrôle des ressources matérielles que pour l'isolation logicielle par des *namespaces*. Tous les conteneurs partagent alors le même noyau, comme sous Linux (cf. figure 1).

Pour fonctionner, un noyau Windows Server est nécessaire, c'est pourquoi ce mode n'est pas disponible sous Windows 10. En revanche, c'est le

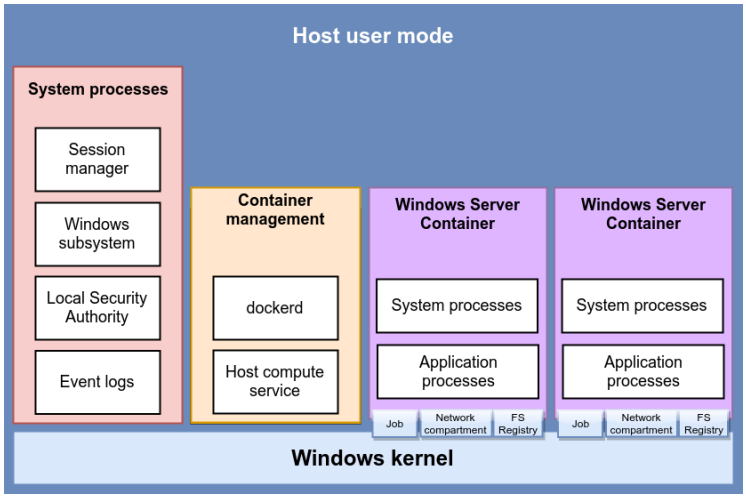

Fig. 1. Mode d'isolation *process* (Windows Server 2016 uniquement)

mode choisi par défaut sous Windows Server 2016, si l'image exécutée est compatible[7]. Pour instancier explicitement un conteneur en mode *process*, il faut passer l'option `--isolation=process` à `docker run`.

Le mode d'isolation d'un conteneur peut être consulté au travers de la commande suivante :

```
PS C:\> docker container inspect -f "{{ .HostConfig.Isolation }}" $ID1
process
PS C:\> docker container inspect -f "{{ .HostConfig.Isolation }}" $ID2
hyperv
```

On notera que le mode d'isolation *process* n'est pas considéré par Microsoft comme étant une frontière de sécurité.

Mode Hyper-V Depuis Windows 10 *Anniversary Update* (version 1607), il est également possible d'instancier des conteneurs Windows depuis la versions cliente du système d'exploitation Microsoft. Cependant, les mécanismes bas niveau requérant un noyau Windows Server, Windows 10 ne supporte que les conteneurs avec isolation Hyper-V (qui lui permet d'instancier une version serveur du noyau).

Les conteneurs Hyper-V sont exécutés dans une machine virtuelle minimaliste, appelée *Utility VM*, et bénéficient ainsi des mécanismes d'isolation de l'hyperviseur Hyper-V. En particulier, ils sont exécutés avec leur propre noyau. Ils peuvent être explicitement instanciés sous Windows Server 2016 au travers de l'option `--isolation=hyperv` du client `docker`.

[7] `https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/version-compatibility`

Sous Windows 10, les conteneurs sont automatiquement exécutés avec cette isolation, le mode `process` n'étant pas disponible (cf. figure 2).

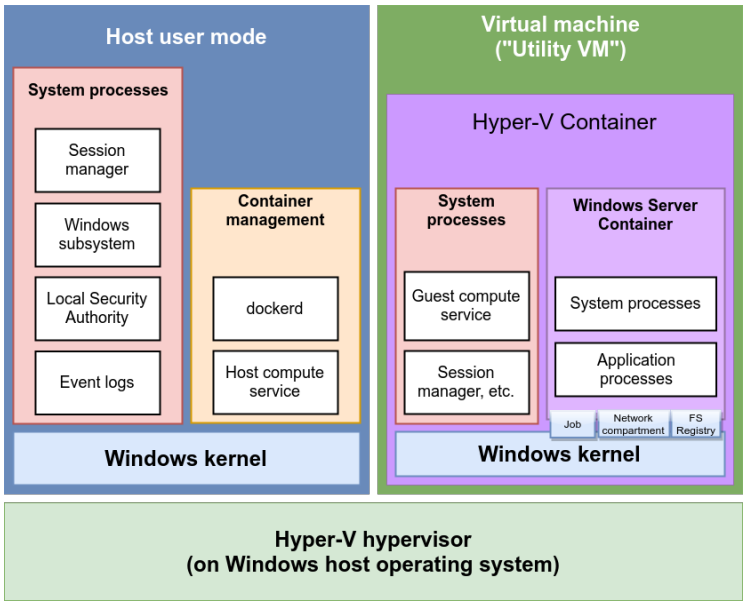

Fig. 2. Mode d'isolation *hyperv* (Windows Server 2016 et Windows 10)

Le système de fichiers du conteneur (ou de l'hôte) peut être partagé grâce à *VMBus*, déjà utilisé par les machines virtuelles Hyper-V, et les communications réseau passent par le mécanisme d'interface réseau virtuelle (*Virtual NIC*) aussi mis en place par Hyper-V.

6.2 Mécanismes d'isolation

Les mécanismes d'isolation à la base des *Windows Server Containers* sont appellés les *server silos* (abrégés par la suite *silos*), une extension de l'objet *Job*. Ce dernier représente un groupe de processus exécutés avec des propriétés et contraintes de ressources communes, à l'instar des *cgroups* sous Linux. On s'intéresse ici aux extensions spécifiques aux silos qui restreignent la vue du système hôte.

Espace de nommage des objets noyau L'espace de nommage des objets (*Object namespace*) est la représentation arborescente des nombreux objets de tous types maintenus par l'*Object Manager* du noyau. Par exemple, la lettre de lecteur `C:` est représentée par l'objet `\GLOBAL??\C:`

de type *SymbolicLink* (pointant vers l'objet `\Device\HarddiskVolumeN` où N est le numéro du volume NTFS).

Dans un conteneur, la vue de cette arborescence est restreinte via la propriété `SiloRootDirectoryName` de l'objet silo associé (à la façon d'un `chroot` sous Linux) : seule une sous-arborescence contenue dans un sous-dossier de `\Silo\` est visible. Cette sous-arborescence est visible depuis l'hôte via des outils comme WinObj de la suite SysInternals (voir figure 3). Si l'on reprend l'exemple des lettres de lecteurs, l'objet `\GLOBAL??\C:` vu depuis un conteneur aura pour nom `\Silo\<silo_id>\GLOBAL??\C:` vu depuis l'hôte (où `<silo_id>` est l'identifiant numérique du Job du conteneur), qui sera un lien symbolique vers `\Silo\<silo_id>\Device\HarddiskVolumeN` (où N sera l'identifiant numérique d'un volume NTFS créé spécifiquement pour le conteneur).

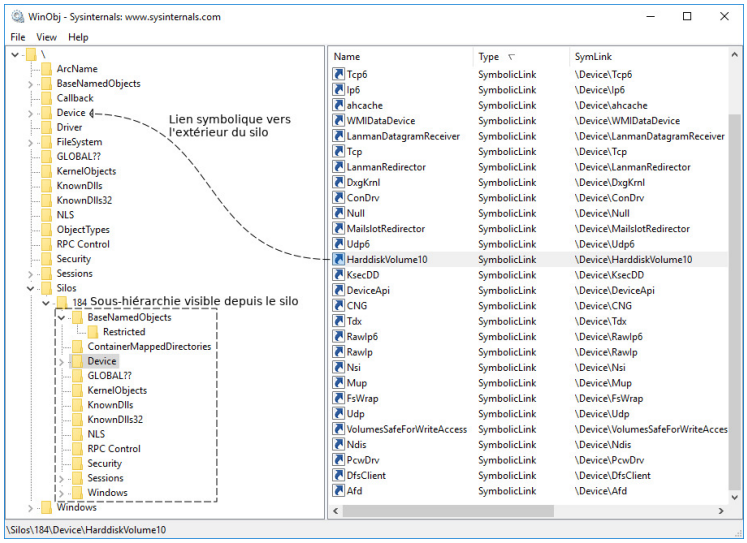

Fig. 3. Virtualisation de l'*Object namespace*

Les objets de l'hôte que le conteneur doit réutiliser (`\Registry` ou `\Device\Tcp` par exemple) sont des liens symboliques pointant vers leur homonyme *à l'extérieur du conteneur*. Les liens symboliques sont par défaut résolus par rapport au `SiloRootDirectoryName` : cette différence sémantique est implémentée sous la forme d'un drapeau `Global` non documenté dans l'objet `SymbolicLink`, indiquant qu'il doit être résolu dans le contexte de l'hôte et non celui du silo. Ce drapeau n'est évidemment pas visible depuis l'intérieur du silo, mais l'est par exemple via WinDBG

si on examine le lien symbolique implémentant un volume monté depuis l'hôte :

```
kd> !object \Silos\104\ContainerMappedDirectories\
Object: ffffaf099fabc080  Type: (ffffc009892269f0) Directory ...
    Hash Address          Type                      Name
    ---- -------          ----                      ----
     33  ffffaf09a1ff33d0 SymbolicLink              4594B0A8-A518-...

kd> !object ffffaf09a1ff33d0
Object: ffffaf09a1ff33d0  Type: (ffffc0098922a3b0) SymbolicLink ...
    Flags: 0x000005 ( Global )
    Target String is '\Device\HarddiskVolume4\Users\Test\Desktop'
```

Processus et threads Contrairement au *PID namespace* fourni par Linux, un silo ne possède pas d'espace d'identifiants de processus (PID) et threads (TID) qui lui soit propre. Le noyau empêche donc seulement un silo d'utiliser ou de faire référence aux identifiants d'un autre silo ou de l'hôte.

Cette approche nécessite de changer la sémantique de tous les appels système manipulant des PID ou TID, pour les rendre conscients de l'existence d'un silo. Les routines bas-niveau du noyau comme `PsLookupProcessById()` et `PsLookupThreadByThreadId()` ont donc été modifiées pour émuler une erreur « *not found* » lorsque le silo du processus demandé est différent du silo en cours.

La limite de cette approche vient du manque de défense en profondeur (le lecteur averti aura remarqué que ce résumé ne mentionne aucune isolation des utilisateurs ou privilèges : il n'y en a pas). En effet, un programme dans un WSC a par défaut des privilèges suffisant pour injecter du code dans un processus hors de son conteneur (voir section 7.4 pour plus de détails), il lui manque seulement une *handle* vers ce dernier. Et, malheureusement, l'appel système `NtGetNextProcess` permet d'en obtenir une sans passer par les routines du noyau prenant en compte les silos (il donne une *handle* vers le processus suivant dans la liste doublement chainée des processus du noyau).

Ce défaut d'isolation démontré par Alex Ionescu lors de la conférence SyScan 360 2017 [4] a été corrigé dans Windows Server à partir de la version 1709, mais persiste dans Windows Server 2016. Il permet une compromission complète de l'hôte par les administrateurs de conteneurs WSC et par tous les utilisateurs du groupe `docker-users`.

Compartiments réseau L'isolation réseau d'un conteneur consiste à lui donner des interfaces réseau, des règles de routage, et des règles de

pare-feu qui lui sont propres. Les *network namespaces* remplissent cette fonction sous Linux, et ont leur équivalent sous Windows, appelés *network compartments*. Dans le réseau NAT par défaut, la première partie d'une interface vNIC émulée (ou *vmNIC synthétique* dans les *Hyper-V containers*) est ajoutée au compartiment de chaque conteneur, et sa contrepartie est connectée à un vSwitch (encore une fois fourni par Hyper-V) qui réalise le lien avec les interfaces physiques de l'hôte.

Le compartiment réseau en cours dans un silo est visible via la commande PowerShell `Get-NetCompartment`, et les interfaces réseau et vSwitchs utilisés sont visibles depuis l'hôte via les commandes `Get-NetAdapter -IncludeHidden` et `Get-VMSwitch` respectivement. Aucune API documentée n'a été trouvée pour lister les règles de pare-feu par compartiment.

7 Audit de sécurité d'un environnement Docker

Pour mener l'audit d'un système, il est d'abord essentiel de définir l'étendue de sa surface d'attaque. Dans le cadre de notre étude d'un environnement Docker, nous allons considérer les éléments suivants :

- le système hôte et les mécanismes bas niveau utilisés par les conteneurs ;
- le gestionnaire de conteneurs et ses interfaces ;
- les conteneurs et leurs interactions avec le monde extérieur ;
- les images, les registres ainsi que les Dockerfiles.

De manière générale, un système hébergeant une brique de l'environnement comme un gestionnaire de conteneurs ou un orchestrateur, se doit d'être minimal. En particulier, seuls les logiciels relatifs aux conteneurs et à leur environnement (stockage, réseau, redondance, etc.) devraient être installés sur l'hôte. De plus, si des informations de sensibilités différentes sont manipulées par plusieurs conteneurs, ces briques et celles les contrôlant devraient toujours être divisées, voire hébergées dans des VMs ou des serveurs différents selon les garanties d'isolation requises.

Afin de mieux appréhender la surface d'attaque il faut comprendre que Docker est architecturé autour de plusieurs composants et en particulier de `libcontainer` évoquée plus haut ainsi que de `runC`, programme exécutant les conteneurs.

- `dockerd` : le démon Docker historique, toujours utilisé comme interface de haut niveau par la commande `docker` ;

- `docker-containerd` : démon « interne » à l'infrastructure docker depuis Docker 1.11, affecté à la gestoin des tâches de bas niveau telles que le stockage, la distribution d'images ou les interfaces réseau ;
- `docker-containerd-ctr` : client de `docker-containerd` ;
- `docker-runc` : programme exécutant les conteneurs eux-même, issu de la standardisation OCI, et réalisant l'interface avec les mécanismes bas niveau d'isolation et de sécurité ;
- `docker-containerd-shim` : glue entre runC et containerd.

7.1 Système hôte

La sécurité du système hôte est centrale dans celle de l'infrastructure hébergeant des conteneurs. Elle est aussi celle qui est la mieux documentée et elle ne fera donc pas l'objet ici d'une explication détaillée, en particulier concernant le durcissement du système d'exploitation.

Certains points nécessitent cependant une attention particulière dans le contexte des environnements exécutant des conteneurs, à savoir :

- la version du système, afin d'en vérifier le maintien en conditions de sécurité mais également le support par l'éditeur ;
- le niveau de mise à jour des outils côté espace utilisateur et en particulier les versions :
  - des bibliothèques et programmes tiers liés aux conteneurs,
  - des gestionnaires de conteneurs, orchestrateurs et registres.

Linux Sous Linux, il est important de vérifier la bonne prise en compte des différents mécanismes d'isolation et de sécurité décrits précédemment. Ceci passe d'abord par la version du noyau en cours d'exécution (par exemple, les *user namespaces* ne sont implémentés qu'à partir du noyau Linux 3.8), mais également par la configuration de celui-ci au travers des options activant les *cgroups*, les *namespaces*, *seccomp* ou encore un système de MAC. On notera que les *capabilities* font partie intégrante du noyau depuis la version 2.6.33 et ne peuvent être désactivées.

- *cgroups* : `CONFIG_MEMCG`, `CONFIG_CPUSETS`, `CONFIG_CGROUPS_PID` ou encore `CONFIG_CGROUP_DEVICE` (liste non exhaustive) ;
- *namespaces* : `CONFIG_UTS_NS`, `CONFIG_IPC_NS`, `CONFIG_USER_NS`, `CONFIG_PID_NS` et `CONFIG_NET_NS`. Les *mount namespaces* et *cgroups namespaces* n'ont pas d'option de configuration dédiée ;
- *seccomp* : `CONFIG_SECCOMP` et `CONFIG_SECCOMP_FILTER` ;

- *capabilities* : elles font partie intégrante du noyau depuis la version 2.6.33 et ne peuvent être désactivées ;
- *MAC* : SELinux ou Apparmor sont respectivement dépendants des options `CONFIG_SECURITY_SELINUX` ou `CONFIG_SECURITY_APPARMOR`.

La plupart des noyaux fournis par les distributions Linux sélectionnent ces options par défaut. Cependant un paramétrage supplémentaire est parfois nécessaire pour activer toutes les fonctions d'isolation, même si celles-ci sont compilées dans le noyau. C'est le cas de Debian où il est nécessaire de positionner le paramètre *sysctl* `kernel.unprivileged_userns_clone` à la valeur 1 pour activer les *user namespaces* (la clé existe aussi sous Ubuntu, mais la valeur est positionnée à 1 par défaut).

Windows Sous Windows, il n'y a pas de paramétrage d'isolation ou de sécurité spécifique à activer sur le système hôte. On se bornera donc à valider la présence des fonctionnalités liées à Hyper-V et aux conteneurs, qui sont de toute manière nécessaires pour utiliser Docker :

```
PS C:\> (Get-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V-All).State
Enabled
PS C:\> (Get-WindowsOptionalFeature -Online -FeatureName Containers).State
Enabled
```

Note : on utilisera `Get-WindowsFeature` sous Windows Server 2016.

7.2 Serveur Docker

Relevé d'informations La commande `docker version` montre si les parties client et serveur de Docker sont tenues à jour (au moins pour les patchs de sécurité). Cette commande indique également si la branche « expérimentale » est déployée sur un hôte :

```
PS C:\> docker version
Client:
 Version:       17.06.2-ee-8-rc1
 API version:   1.30
 Go version:    go1.8.7
 Git commit:    061a8cb
 Built: Fri Mar 23 22:36:03 2018
 OS/Arch:       windows/amd64

Server:
 Engine:
  Version:      17.06.2-ee-8-rc1
  API version:  1.30 (minimum version 1.24)
  Go version:   go1.8.7
  Git commit:   061a8cb
```

```
  Built:        Fri Mar 23 22:38:34 2018
  OS/Arch:      windows/amd64
  Experimental: false
```

La commande `docker info` donne un premier état des lieux de l'installation de Docker sur le système hôte et de la configuration de certains paramètres de sécurité :

```
$ docker info
[...]
Server Version: 1.13.1
[...]
Logging Driver: journald
Cgroup Driver: systemd
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Authorization: rhel-push-plugin
Swarm: inactive
Runtimes: oci runc
Default Runtime: oci
Init Binary: /usr/libexec/docker/docker-init-current
containerd version:  (expected: aa8187dbd3b7ad67d8e5e3a15115d3eef43a7ed1)
runc version: N/A (expected: 9df8b306d01f59d3a8029be411de015b7304dd8f)
init version: N/A (expected: 949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
 seccomp
  WARNING: You're not using the default seccomp profile
  Profile: /etc/docker/seccomp.json
selinux
Kernel Version: 4.13.5-200.fc26.x86_64
Operating System: Fedora 26 (Twenty Six)
[...]
Insecure Registries:
127.0.0.0/8
Registries: docker.io (secure), registry.fedoraproject.org (secure),
            registry.access.redhat.com (secure)
```

On prêtera attention aux registres autorisés, en priorité aux registres sans protection TLS ou associés à un certificat auto-signé ou d'une autorité non reconnue (catégorie « *insecure registries* »). Les éventuelles solutions de MAC reconnues devront aussi être inspectées (ici, les règles SELinux). Le profil *seccomp* appliqué par défaut est également référencé, et doit être vérifié s'il a été personnalisé comme c'est le cas ici.

Liste des conteneurs La liste des conteneurs en cours d'exécution sur le système hôte est obtenue par les commandes `docker ps` ou `docker container ls` :

```
# docker ps
CONTAINER ID  IMAGE       [...] STATUS      PORTS
```

```
de849b4527ea  registry:2        Up 2 hours  0.0.0.0:5000->5000/tcp
1b16ff98eb1c  ubuntu           Up 2 hours
e26e89319c23  eed8c09818d4      Up 4 hours  0.0.0.0:80->80/tcp
```

À chaque conteneur est associé un identifiant unique (`CONTAINER ID`) codé sur 256 bits et qui servira de référence pour de nombreuses autres commandes. Pour obtenir la version longue de cet identifiant, il est possible de passer l'option `--no-trunc` à la commande précédente.

La liste des conteneurs dont l'exécution est terminée, mais qui n'ont pas été détruits, est disponible en passant l'option `--all`.

Interfaces d'administration Le démon `dockerd` peut être configuré pour recevoir des ordres sur plusieurs interfaces à la fois : sockets TCP, et sockets UNIX sur Linux ou canaux nommés sur Windows. Chaque interface, entre de mauvaises mains, permet la compromission de l'hôte [5], elles doivent donc répondre à un impératif métier et être protégées. Sous Linux, la communication entre les clients et le démon Docker est faite par défaut au travers de la socket Unix `/var/run/docker.sock`, dont on vérifiera qu'elle n'est accessible que par `root` et les membres du groupe `docker` :

```
$ ls -l /var/run/docker.sock
srw-rw---- 1 root docker 0 Jan 26 22:58 /var/run/docker.sock
$ getent group docker
```

Sous Windows, l'accès au service Docker se fait par des canaux nommés :

- `\\.\pipe\docker_engine` pour les conteneurs Linux ;
- `\\.\pipe\docker_engine_windows` pour les conteneurs Windows.

Par défaut, ces canaux ne sont accessibles que par les membres du groupe intégré `Administrators` et ceux du groupe local `docker-users` :

```
PS C:\> [System.IO.Directory]::GetAccessControl("\\.\pipe\docker_engine") | fl
Path   :
Owner  : BUILTIN\Administrators
Group  : DOCKER-W2016\None
Access : NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         DOCKER-W2016\docker-users Allow  Write, Read, Synchronize
Audit  :
Sddl   : O:BAG:S-1-5-21-3698178738-2750322818-760387437-513D:P(A;;FA;;;SY)(A;;FA;;;BA)
         (A;;0x12019f;;;S-1-5-21-3698178738-2750322818-760387437-1000)
```

Les groupes `docker` sous Linux et `docker-users` sous Windows ont les mêmes limitations en matière de sécurité, à savoir qu'en être membre revient à disposer des plus hauts privilèges sur le système [5]. Il est donc important de vérifier que seuls les utilisateurs de confiance nécessaires en font partie.

Une ou plusieurs interfaces d'administration réseau (TCP) peuvent être rajoutées, par exemple avec l'option `-H tcp://0.0.0.0:2375`. Ces dernières ne demandent par défaut aucune authentification, on veillera dans ce cas à ce qu'elles soient protégées par un *reverse-proxy* authentifiant, ou au moyen de l'option `--tlsverify` de `dockerd` qui force chaque client à présenter un certificat X.509 valide. Dans ce cas on examinera comment sont générés et stockés les certificats clients et leurs clés privées. En effet, certaines solutions « clé en main » (scripts PowerShell[8] ou images Docker[9]) configurent Docker avec une autorité racine, mais laissent la clé privée de l'autorité de certification sur disque sans raison, protégée par un mot de passe prédictible ou stocké aux côtés du fichier chiffré.

Gestion des autorisations Par défaut, avoir accès à l'une des interfaces d'administration, par exemple après authentification avec certificat client sur l'interface TCP, donne accès à l'ensemble des fonctions exposées par l'API Docker, sans limitation.

Docker permet l'inclusion de *plugins*[10] chargés de valider utilisateur par utilisateur les autorisations d'accès à son API. Ces derniers se chargent d'intercepter les requêtes REST effectuées par les clients et appliquent une politique de sécurité définie par l'administrateur afin d'autoriser ou de refuser la demande. La société Twistlock a été la première à fournir une implémentation de ce type de *plugin*[11] et bien que cette dernière soit très simple et ne permette pas de filtrage fin, elle reste néanmoins suffisante pour de nombreux usages.

Ces *plugins* sont activés au travers de l'option `--authorization-plugin` de `dockerd`. On s'intéressera en particulier à la politique de sécurité ainsi définie et aux fonctionnalités dangereuses qui pourraient être filtrées, comme la possibilité de créer des conteneurs privilégiés ou des volumes partageant des répertoires depuis l'hôte dans un conteneur.

[8] `https://github.com/MicrosoftDocs/Virtualization-Documentation/tree/master/windows-server-container-tools/DockerTLS`
[9] `https://hub.docker.com/r/stefanscherer/dockertls-windows/`
[10] `https://docs.docker.com/engine/extend/plugins_authorization/`
[11] `https://github.com/twistlock/authz`

Il conviendra aussi de faire attention à l'ordre des politiques appliquées et spécialement à la politique par défaut (celle liée à l'utilisateur « vide ») qui s'applique également aux connexions sur la socket Unix (ou sur le canal nommé sous Windows).

Journalisation Docker journalise de nombreux événements qu'il peut être intéressant de récupérer dans le cadre d'un audit de sécurité.

Les journaux du gestionnaire de conteneurs sont stockés par défaut dans un fichier dont la localisation dépend du système hôte. Sous Debian, il s'agit de `/var/log/daemon.log`. Sous Windows, il est possible de consulter les journaux avec `Get-Eventlog -LogName Application -Source docker` (journal Applications de Windows) ou dans le répertoire `%USERPROFILE%\AppData\Local\Docker` pour chaque utilisateur.

Par ailleurs, Docker considère que les journaux d'une application lancée dans un conteneur sont constitués des deux flux `stdout` et `stderr`. C'est pourquoi la documentation recommande d'y exécuter les serveurs d'application en premier plan afin de capturer ces flux. Ces journaux sont accessibles au travers des commandes `docker logs CONTAINERID`.

Depuis Docker 17.05, il est possible de préciser la méthode de stockage des journaux au travers d'un *logging driver*[12] directement dans le fichier de configuration `daemon.json`. Windows utilise le *driver* `json-file` par défaut, stockant les journaux dans des fichiers, alors que Fedora est configuré avec le *driver* `syslog` ou `journald`. Voici un exemple de journalisation vers un server syslog distant :

```
{
  "log-driver": "syslog",
  "log-opts": {
    "syslog-address": "udp://10.2.5.10:514"
  }
}
```

Les événements du service `dockerd` ne sont par défaut pas conservés sur le long terme, on s'intéressera donc au *logging driver* et à l'emplacement de stockage des journaux spécifiques à chaque conteneur :

```
$ docker inspect -f '{{.HostConfig.LogConfig.Type}}' CONTAINERID
journald
$ journalctl --all -b CONTAINER_NAME=CONTAINERID
Apr 02 14:13:51 testhost dockerd-current[19024]: / # echo Hello, Docker
```

Ou par exemple, sous Windows :

```
PS C:\> docker inspect -f "{{ .LogPath }}" CONTAINERID
C:\ProgramData\Docker\containers\CONTAINERID\CONTAINERID-json.log
```

[12] `https://docs.docker.com/v17.09/engine/admin/logging/overview/`

7.3 Configuration réseau

À mi-chemin entre la configuration de l'hôte et du conteneur, plusieurs types de réseaux sont instanciables sous Linux et Windows.

Le plus simple et sécurisant si le conteneur n'a pas besoin de connectivité réseau est `none`, qui n'utilise aucun *driver* car il ne fournit qu'une interface de `loopback` spécifique au conteneur. On notera que cette dernière ne permet pas non plus de contacter les services écoutant localement à l'hôte, grâce au mécanisme de *network namespaces* sous Linux ou de *network compartments* sous Windows.

Sous Linux, le réseau utilisé par défaut pour tous les nouveaux conteneurs est de type `bridge`, basé sur un bridge virtuel fourni par le noyau et une paire d'interfaces Ethernet virtuelles (une pour le namespace du conteneur, une pour l'hôte). L'équivalent sous Windows est le réseau `nat`, basé sur des *vSwitch* et des *vNIC* fournis par Hyper-V. Hormis dans le cas du bridge par défaut sous Linux, ce type de réseau conduit le service `dockerd` à écouter sur les ports TCP et UDP 53 pour fournir un serveur DNS aux conteneurs. On notera que ceci rajoute sous Windows une règle de pare-feu laissant entrer ce traffic pour n'importe quel exécutable, que Docker soit actif ou non. Dans cette configuration, les conteneurs peuvent par défaut se joindre sans passer par la couche IP de l'hôte, ce qui ouvre la voie aux attaques par empoisonnement de cache ARP ou DNS, et expose à chacun tous les services des autres, sans que cela ne réponde toujours à un besoin fonctionnel. On préfèrera utiliser des bridges dédiés par classes de services ayant besoin de communiquer, ou à défaut reconfigurés pour limiter la connectivité au niveau TCP/UDP aux seuls services explicitement exposés (`com.docker.network.bridge.enable_icc: false`, visible via la commande `docker network inspect`).

L'association de chaque vNIC à son vSwitch est accessible en PowerShell, ainsi que l'état des fonctions de protection intégrées à Hyper-V :

```
PS C:\Administrator> Get-VMNetworkAdapter -All | fl *

Name               : Container Port 9417d7ef # Interface virtuelle du conteneur
MacAddress         : 00155D34099E
SwitchName         : Layered Local Area Connection* 1 # vSwitch assigné
AclList            : {}  # Restrictions de source, destination IP
ExtendedAclList    : {}  # Restrictions de protocoles et ports
MacAddressSpoofing : Off # Autorisation d'usurper des adresses MAC
DhcpGuard          : Off # Interdiction d'émettre des trames serveur DHCP
RouterGuard        : Off # Interdiction d'émettre des Router Advertisements IPv6
StormLimit         : 0   # Nombre de paquets broadcast, multicast ou
                         #inconnus autorisés par seconde, ici sans limite
...
```

Le mode *hôte* sous Linux laisse le conteneur dans le *namespace* réseau de son hôte, et doit donc être réservé aux images de confiance et aux conteneurs sans risque de compromission : en effet, ces derniers pourraient modifier la configuration de routage et les règles de pare-feu de l'hôte. L'équivalent sous Windows est le mode *transparent*, qui donne au conteneur un compartiment réseau, et une vNIC placée dans un bridge commun avec une interface physique. Bien qu'avantageux par rapport à Linux en matière de restriction de privilège, ce mode ne devrait être utilisé que sur des réseaux de confiance en l'état actuel car le conteneur est alors directement exposé sur le réseau, or les conteneurs Docker pour Windows ont leur service de pare-feu totalement désactivé par défaut.

Pour finir, quand Docker est exécuté en mode *Swarm*, il joint un groupe de nœuds collaborant pour exécuter des conteneurs selon une politique potentiellement complexe de redondance. Idéalement, deux adresses différentes devraient être utilisées par chaque nœud pour segmenter le trafic de contrôle et le trafic applicatif (on veillera à ce que `--advertise-addr` et `--datapath-addr` aient été renseignés à l'initialisation du Swarm). Dans ce mode, les réseaux de type *superposition* (*overlay*) sont visibles sur tous les nœuds du Swarm, et connectent les conteneurs associés même s'ils sont exécutés par des nœuds différents. Deux réseaux particuliers `ingress` et `docker_gwbridge` sont créés par défaut, respectivement pour acheminer le trafic de gestion du Swarm et pour connecter les démons Docker. Sous Linux, ce trafic est par défaut chiffré avant d'être encapsulé, contrairement au trafic applicatif. On veillera donc à ce qu'un réseau de superposition avec l'option `--opt encrypted` soit utilisé. Les hôtes Windows ne devraient pour le moment pas être utilisés en mode Swarm, à moins que seuls des protocoles réseau sécurisés soient utilisés, ou que le réseau interconnectant les nœuds soit de confiance, car Docker pour Windows ne supporte pas le chiffrement des réseaux de superposition.

7.4 Environnement d'exécution des conteneurs

Nom d'ordinateur A l'instar de l'*UTS namespace* de Linux, chaque conteneur Windows dispose de son propre nom d'ordinateur, dérivé de l'identifiant unique créé par docker. Sur l'hôte, notons le `CONTAINER ID` fourni par docker :

```
PS C:\> $env:ComputerName
DOCKER-W2016
PS C:\> docker ps
CONTAINER ID  IMAGE                         COMMAND           CREATED       STATUS
ddcf55000cdf  microsoft/windowsservercore  "powershell.exe"  2 hours ago   Up 2 hours
```

On le retrouve dans le conteneur comme nom d'ordinateur :

```
[CONTAINER] PS C:\Users\ContainerUser> $env:ComputerName
DDCF55000CDF
```

Prise d'informations

Informations générales La commande `docker inspect` donne de nombreuses informations sur chaque instance de conteneur et en particulier le PID du premier processus (ici 7509) :

```
# docker inspect CONTAINERID
[
  {
    "Id": "CONTAINERID",
    "Created": "2018-01-27T19:52:30.964550224Z",
    "Path": "/entrypoint.sh",
    "Args": [
      "/etc/docker/registry/config.yml"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 7509,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2018-01-27T19:52:31.256721757Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image": "sha256:d1fd7d86[...]",
 [...]
]
```

Cette commande est centrale pour comprendre l'environnement de chaque conteneur et permettra de croiser les informations avec les relevés suivants pour valider la mise en place des mécanismes d'isolation et de sécurité.

Sous Windows, on pourra s'assurer qu'aucun objet, volume ou clé de registre n'a été partagé avec le conteneur en comparant la liste des objets partagés par défaut (dans `C:\Windows\System32\containers\wsc.def`) à celle d'une installation inchangée. On pourra aussi inspecter le contenu d'un silo en cours d'exécution depuis l'extérieur via le programme `WinObj` de la suite *Sysinternals*, qui permet de naviguer dans l'*Object namespace* (cf. figure 3). S'il n'est pas possible d'installer d'outils sur l'hôte, on pourra

faire de même depuis l'intérieur du conteneur mais en ligne de commande uniquement, via l'outil `lsobj`[13] développé pour ce cas d'usage.

Processus et services La commande `docker top` donne la liste des processus exécutés à l'intérieur d'un conteneur donné, avec leur PID *sur le système hôte* :

```
C:\>docker top CONTAINERID1
Name            PID         CPU                 Private Working Set
smss.exe        880         00:00:00.156        221.2kB
csrss.exe       904         00:00:00.437        352.3kB
wininit.exe     948         00:00:00.031        643.1kB
[...]
svchost.exe     800         00:00:00.078        1.36MB
CExecSvc.exe    1400        00:00:00.031        716.8kB
svchost.exe     1408        00:00:00.203        2.29MB

# docker top CONTAINERID2
UID     PID     PPID    C    STIME   TTY     TIME       CMD
root    15571   15557   0    00:02   pts/0   00:00:00   /bin/sh
root    15738   15651   0    00:15   pts/1   00:00:00   ping www.ssi.gouv.fr
```

Ressources matérielles Docker permet d'empêcher un conteneur d'accaparer les ressources matérielles (mémoire physique, temps d'exécution sur le processeur, ou encore bande passante d'accès réseau ou disque), ce qui entrainerait un déni de service de l'hôte, intentionnel ou non. Sous Linux la restriction est appliquée par l'intermédiaire des *cgroups*[14], et sous Windows par les limites intégrées aux objets Jobs[15], certaines depuis Windows XP. On regardera donc par exemple si les options `--cpus`, `--memory`, `--io-maxbandwidth`, `--io-maxiops` ou encore `--pids-limit` ont été utilisées lors de la création du conteneur, par la commande `docker inspect`. Sur Windows, on pourra aussi directement lire ces limites dans les options du job, par exemple avec *Process Explorer* de la suite *sysinternals* (cf. figure 4).

Durcissement Linux

Restriction d'accès aux périphériques Le contrôleur *cgroup* `devcg` peut restreindre l'accès à certains périphériques au travers de listes blanches et/ou de listes noires. Par défaut[16], les pseudo-périphériques de base

[13] https://github.com/mtth-bfft/lsobj
[14] https://docs.docker.com/config/containers/resource_constraints/
[15] https://docs.microsoft.com/fr-fr/virtualization/windowscontainers/manage-containers/resource-controls
[16] https://github.com/opencontainers/runtime-spec/blob/master/config-linux.md#default-devices

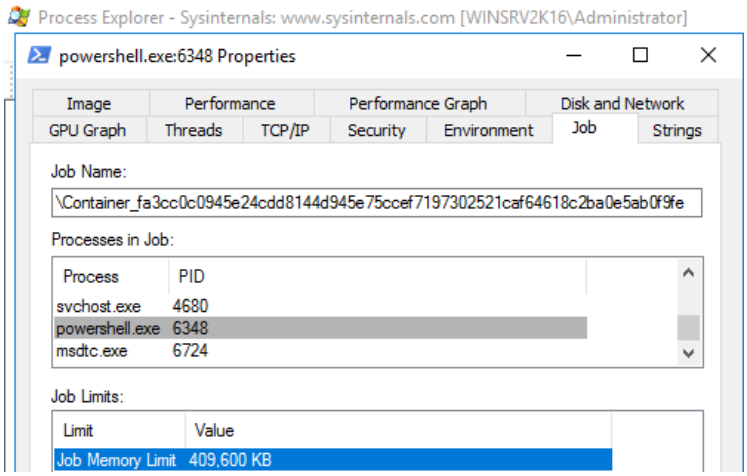

Fig. 4. Visualisation des limitations de ressources pour un Job

nécessaires au démarrage du conteneur sont accessibles (ceci inclut notamment `/dev/null`, `/dev/random` ou encore `/dev/console`). On s'assurera que les éventuels autres périphériques rajoutés par `--device` ou `--device-cgroup-rule` ne mettent pas à mal l'isolation entre le conteneur et l'hôte (par exemple en partageant le disque dur contenant l'OS de l'hôte, permettant au conteneur de modifier son système de fichier et piéger des exécutables), et que les permissions appliquées sont correctes (lecture, écriture, ou création en fonction du besoin) :

```
# docker run -itd --device-cgroup-rule "b 8:0 rwm" \
                  --device /dev/ttyUSB0:/dev/ttyUSB0:r alpine
# docker inspect -f "{{ .HostConfig.DeviceCgroupRules }}" CONTAINERID
[b 8:0 rwm]   # 8:0 correspond ici à /dev/sda
# docker inspect -f "{{ .HostConfig.Devices  }}" CONTAINERID
[{/dev/ttyUSB0 /dev/ttyUSB0 r}]
```

Espaces de nommage Sous Linux, il est possible de vérifier, conteneur par conteneur, si l'utilisation des *namespaces* est correcte au moyen de `docker inspect` :

```
# docker inspect CONTAINERID | grep -iE "(network|ipc|pid|uts)mode"
          "NetworkMode": "default",
          "IpcMode": "shareable",
          "PidMode": "",
          "UTSMode": "",
```

On s'intéressera à vérifier que les options ci-dessus ne sont pas positionnées à la valeur `host` (par exemple en utilisant `--ipc=host`) indiquant une désactivation des *namespaces* associés. Par défaut, ces options ont

une valeur acceptable et fournissent une isolation correcte, notamment une valeur nulle indique que le processus est dans son propre *namespace*.

La signification des valeurs des différentes options est disponible sur la page de la documentation consacrée à `docker run`[17].

Pour plus de détails sur chaque processus, la commande `lsns` permet de lister les *namespaces* et les processus qui y sont rattachés en lui donnant un PID :

```
# lsns -p $(docker inspect -f "{{ .State.Pid }}" CONTAINERID)
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    106     1 root /sbin/init
4026531837 user      106     1 root /sbin/init
4026532751 mnt         3 15571 root /bin/sh
4026532752 uts         3 15571 root /bin/sh
4026532753 ipc         3 15571 root /bin/sh
4026532754 pid         3 15571 root /bin/sh
4026532756 net         3 15571 root /bin/sh
```

On retrouve, vu du système hôte, le PID du processus exécuté dans le conteneur (ici 15571), les *namespaces* associés et leurs identifiants, ainsi que les utilisateurs exécutant les différents processus. Il est ainsi possible de vérifier l'isolation effective des processus exécutés par le conteneur. Dans l'exemple ci-dessus, on remarque que les *namespaces cgroup* et *user* sont rattachés au PID 1, indiquant que le conteneur n'en possède pas une instance dédiée.

User namespaces Le cas des *user namespaces* est à considérer à part. L'activation de cette fonctionnalité doit être faite de façon globale au niveau du démon `dockerd` par l'option `userns-remap`[18], qui nécessite la création d'un utilisateur du système hôte vers lequel sera réaffecté l'UID 0 de `root`. Cet utilisateur sera créé automatiquement si l'on passe la valeur `userns=default`, ce qu'il convient de vérifier ainsi :

```
# grep userns-remap /etc/docker/daemon.js
  "userns-remap": "default"
# getent passwd dockremap
dockremap:x:108:112::/home/dockremap:/bin/false
# grep dockremap /etc/subuid
dockremap:231072:65536
```

Après avoir relancé `dockerd`, on constate que les nouveaux conteneurs ainsi exécutés utilisent les *user namespaces* et que l'utilisateur root a bien été remplacé par l'utilisateur à l'UID 231072 :

[17] `https://docs.docker.com/engine/reference/run/`
[18] `https://docs.docker.com/engine/security/userns-remap/`

```
# lsns -p $(docker inspect -f "{{ .State.Pid }}" CONTAINERID)
        NS TYPE   NPROCS   PID USER   COMMAND
4026531835 cgroup    100     1 root   /sbin/init
4026532432 user        1 19478 231072 sh
4026532433 mnt         1 19478 231072 sh
[...]
```

On vérifiera donc que l'option `userns-remap` est bien activée afin que cette réaffectation soit effective pour tous les conteneurs. On s'assurera également que l'option `--userns` n'a pas été placée à la valeur `host`, qui désactiverait le mécanisme d'isolation pour un conteneur donné :

```
# docker inspect b490806f48e | grep -E UsernsMode
            "UsernsMode": "",
```

Capabilities Par défaut, Docker donne un ensemble restreint de *capabilities* au PID 1 de chaque conteneur. Qui plus est, les conteneurs sont définitivement privés de toute autre *capability* par le biais de leur *bounding set* :

```
# grep -i cap /proc/$PID/status
CapInh:         00000000a80425fb
CapPrm:         00000000a80425fb
CapEff:         00000000a80425fb
CapBnd:         00000000a80425fb
CapAmb:         0000000000000000
```

Ici, on voit que toutes les *capabilities* du *bounding set* forcé par Docker sont activées par le processus inspecté. Si cela n'avait pas été le cas, on aurait pu s'intéresser aux *capabilities* de l'ensemble *permitted*, puis voir si certaines auraient permis de regagner des capabilities, et aux exécutables accessibles depuis le conteneur qui auraient pu conférer des *capabilities* supplémentaires, une fois exécutés. L'outil `capsh` du paquet `libcap2-bin` permet de décoder les ensembles de *capabilities* sous forme numérique :

```
# capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,
cap_mknod,cap_audit_write,cap_setfcap
```

Il est possible d'agrémenter ou de restreindre cette liste au lancement d'un nouveau conteneur par les options `--cap-add` et `--cap-drop`. Il convient alors de vérifier si ces options ont été utilisées afin de comprendre les droits rajoutés ou supprimés pour le conteneur :

```
# docker inspect -f "{{ .HostConfig.CapDrop }}" CONTAINERID
[CHMOD]
# docker inspect -f "{{ .HostConfig.CapAdd}}" CONTAINERID
[SYS_ADMIN]
```

Seccomp Le mode *seccomp* activé pour un processus est également exposé par le noyau dans `/proc` :

```
# grep -i seccomp /proc/$PID/status
Seccomp:  2
```

Ici `Seccomp: 2` indique que nous avons affaire à un filtrage BPF, un programme écrit dans une version restreinte du langage cBPF[19]. Les filtres mis en œuvre peuvent être récupérés depuis Linux 4.4 en s'attachant au processus via `ptrace()` et en utilisant l'option `PTRACE_SECCOMP_GET_FILTER`. Il est de plus toujours possible de récupérer le profil *seccomp* chargé par Docker au démarrage du serveur, à comparer à sa version par défaut disponible sur le Github du projet Moby[20] :

```
# docker info | grep Profile
  Profile: /etc/docker/seccomp.json
```

Néanmoins, plusieurs options peuvent altérer le profil *seccomp* par défaut, comme les options modifiant les *capabilities* ou `--privileged`. On pourra donc, en cas de doute, copier le filtre BPF brut depuis le noyau et en déduire les appels système qu'il autorise en lui appliquant des méthodes formelles. Ici, nous avons créé une implémentation de la sémantique cBPF en langage Prolog comme preuve de concept[21]. Une fois le programme copié depuis le noyau et désassemblé, on obtient ainsi une base de connaissances Prolog :

```
# seccomp-dump -p $PID > docker.pl
$ swipl ./seccomp.pl
?- [docker].
```

On peut alors interroger cette base, par exemple pour lui demander si l'appel système `init_module()` est autorisé sur l'architecture `x86_64` (heureusement, la réponse est non) :

```
?- filter_accepts(175, 3221225534, Arg1, Arg2, Arg3, Arg4, Arg5, Arg6).
false.
```

Ou, plus exhaustivement, on pourra demander la liste exacte des appels système autorisés, et vérifier qu'aucun ne permet de contourner un mécanisme d'isolation ou accéder à une ressource qui n'est pas isolée (chargement de module noyau, accès aux périphériques, etc.). Ce travail

[19] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/seccomp.c

[20] https://github.com/moby/moby/blob/master/profiles/seccomp/default.json

[21] https://github.com/mtth-bfft/seccomp-analyze

demande encore un effort considérable d'automatisation, étant donné la taille des filtres générés par Docker (de l'ordre de 950 instructions lors de nos essais) :

```
?- filter_accepts(Nr, Arch, Rip, Arg1, Arg2, Arg3, Arg4, Arg5, Arg6), \
   write('Nr='), write(Nr), write(' Arch='), write(Arch), write('\n'), fail.
Nr=66 Arch=1073741827
Nr=70 Arch=1073741827
Nr=71 Arch=1073741827
Nr=75 Arch=1073741827
Nr=76 Arch=1073741827
[...]
```

Conteneur privilégié Docker dispose d'une option `--privileged`, qui désactive les mécanismes de sécurité appliqués aux conteneurs lors de leur instanciation, et en particulier *seccomp*, le filtre des *capabilities*, la limitation d'accès aux périphériques par les *cgroups* et l'application des profils de MAC. Elle correspond aux options de lignes de commande `--cap-add ALL --security-opt apparmor=unconfined --security-opt seccomp=unconfined` (ou `selinux=unconfined` si SELinux est utilisé à la place d'AppArmor).

Cette option est par ailleurs incompatible avec les *user namespaces* et nécessite également de passer `--userns=host` si ces derniers sont activés.

On vérifiera donc à ce que cette option ne soit activée pour aucun conteneur.

Enfin, on peut noter que `docker exec --privileged` ne désactive pas *seccomp*, mais ce comportement pourrait changer dans le futur[22].

Élévation de privilèges Les conteneurs peuvent être instanciés avec l'option de sécurité `no_new_privs` qui va passer le paramètre `PR_SET_NO_NEW_PRIVS` à l'appel système `prctl()`[23] indiquant que les processus ne pourront par exemple pas profiter des bits setuid, setgid, ni des capabilities possédées par des fichiers pour élever leurs privilèges (ils ne pourront donc pas non plus utiliser des programmes comme `su` ou `sudo`). Cette option peut être passée à `docker run` sous la forme `--security-opt no-new-privileges`.

On peut vérifier le paramétrage de cette option avec :

```
$ docker inspect -f "{{ .HostConfig.SecurityOpt }}" CONTAINERID
[no-new-privileges]
```

[22] `https://github.com/docker/docker/issues/2198`
[23] `https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt`

Durcissement Windows Windows ne fournit pas de mécanisme d'abstraction des identifiants des utilisateurs, comme Linux au travers des *user namespaces*. La plupart des processus système du conteneur sont exécutés avec les droits de l'entité `LocalSystem` de l'hôte. Seul le processus utilisateur défini dans le fichier *DockerFile* (directives `CMD` ou `ENTRYPOINT`), ou issu de la ligne de commande, sera exécuté avec un autre utilisateur, soit au travers d'une directive `USER`, soit en utilisant l'option `--user` de docker (cf. figure 5, ne fonctionne qu'à partir de Windows 1703).

Fig. 5. Processus exécutés par un *Windows Server Container* (Windows Server 2016)

Si l'on regarde les processus exécutés dans le conteneur, on retrouve les mêmes informations et notamment les identifiants de processus identiques entre l'intérieur et l'extérieur du conteneur :

```
[CONTAINER] PS C:\> $owners = @{}
[CONTAINER] PS C:\> gwmi win32_process | % { $owners[$_.handle] = $_.getowner().user }
[CONTAINER] PS C:\> Get-Process | Select ProcessName, Id, \
                            @{l="Owner";e={$owners[$_.id.tostring()]}}

ProcessName   Id Owner
-----------   -- -----
CExecSvc    2640 SYSTEM
csrss       2724 SYSTEM
lsass       3552 SYSTEM
powershell  4308 ContainerAdministrator
services    2248 SYSTEM
smss        2012 SYSTEM
wininit     3204 SYSTEM
WmiPrvSE    2396 SYSTEM
[...]
```

Le processus `CExecSvc` est chargé de l'exécution de la tâche utilisateur à l'intérieur du conteneur, avec l'identité `ContainerAdministrator` par défaut.

```
[CONTAINER] PS C:\> whoami
user manager\containeradministrator
[CONTAINER] PS C:\> [Security.Principal.WindowsIdentity]::GetCurrent() | fl *
User   : S-1-5-93-2-1
[...]
```

Utilisateurs et privilèges On a vu que les utilisateurs à l'intérieur d'un conteneur sous Linux peuvent soit avoir les mêmes droits que sur l'hôte, soit être réassignés à des UID différents (par exemple, un utilisateur `root` d'un conteneur n'est pas forcément `root` sur l'hôte en cas d'évasion du conteneur). Sous Windows, il n'existe pas d'équivalent aux *user namespaces*.

Dans les images de conteneurs Windows fournies par Microsoft, et en particulier ici `microsoft/windowsservercore`, aucun utilisateur local n'existe, en dehors des trois créés par défaut :

```
[CONTAINER] PS C:\> Get-LocalUser

Name           Enabled Description
----           ------- -----------
Administrator  False   Built-in account for administering the computer/domain
DefaultAccount False   A user account managed by the system.
Guest          False   Built-in account for guest access to the computer/domain
```

De plus, la commande exécutée par l'image a l'identité `User Manager\ContainerAdministrator` (SID `S-1-5-93-2-1`), identique dans tous les conteneurs. À ce jour, ce SID n'est présent dans aucune documentation officielle de Microsoft, ne fait pas partie de la liste des *Well-Known SIDs*, et n'est pas résolu sur l'hôte :

```
PS C:\> PsGetsid.exe -nobanner "user manager\containeradministrator"
Error querying account:
No mapping between account names and security IDs was done.
```

La commande `whoami /all` montre les groupes auxquels appartient `ContainerAdministrator`. On remarque notamment que cet utilisateur est membre de `BUILTIN\Administrators` :

```
[CONTAINER] PS C:\> whoami /user /groups
[...]
User Name                              SID
====================================== ============
user manager\containeradministrator S-1-5-93-2-1

[...]
Group Name                             Type             SID
====================================== ================ ============
Mandatory Label\High Mandatory Level Label            S-1-16-12288
Everyone                               Well-known group S-1-1-0
BUILTIN\Users                          Alias            S-1-5-32-545
NT AUTHORITY\SERVICE                   Well-known group S-1-5-6
CONSOLE LOGON                          Well-known group S-1-2-1
NT AUTHORITY\Authenticated Users       Well-known group S-1-5-11
NT AUTHORITY\This Organization         Well-known group S-1-5-15
LOCAL                                  Well-known group S-1-2-0
BUILTIN\Administrators                 Alias            S-1-5-32-544
                                       Unknown SID type S-1-5-93-0
```

Par ailleurs, cet utilisateur dispose de nombreux privilèges dont `SeDebugPrivilege` activé par défaut :

```
[CONTAINER] PS C:\> whoami /priv

Privilege Name            Description                               State
========================= ========================================= ========
[...]
SeTakeOwnershipPrivilege  Take ownership of files or other objects  Disabled
SeLoadDriverPrivilege     Load and unload device drivers            Disabled
[...]
SeBackupPrivilege         Back up files and directories             Disabled
SeRestorePrivilege        Restore files and directories             Disabled
SeShutdownPrivilege       Shut down the system                      Disabled
SeDebugPrivilege          Debug programs                            Enabled
[...]
SeImpersonatePrivilege    Impersonate a client after authentication  Enabled
SeCreateGlobalPrivilege   Create global objects                      Enabled
[...]
```

Une autre entité nommée `ContainerUser`, est également présente dans le conteneur, avec le SID `S-1-5-93-2-2`. On peut récupérer les mêmes informations pour cette entité, à propos de ses groupes et privilèges :

```
PS C:\Users\Administrator> Enter-PSSession -ContainerId CONTAINERID
[CONTAINERID]: PS C:\Users\ContainerUser\Documents> whoami /all
[...]
User Name                  SID
========================== ============
user manager\containeruser S-1-5-93-2-2

[...]
Group Name                                 Type             SID
========================================== ================ ============
Mandatory Label\High Mandatory Level       Label            S-1-16-12288
Everyone                                   Well-known group S-1-1-0
BUILTIN\Users                              Alias            S-1-5-32-545
NT AUTHORITY\SERVICE                       Well-known group S-1-5-6
CONSOLE LOGON                              Well-known group S-1-2-1
NT AUTHORITY\Authenticated Users           Well-known group S-1-5-11
NT AUTHORITY\This Organization             Well-known group S-1-5-15
LOCAL                                      Well-known group S-1-2-0
                                           Unknown SID type S-1-5-93-0
[...]
Privilege Name                Description                               State
============================= ========================================= ========
SeChangeNotifyPrivilege       Bypass traverse checking                  Enabled
SeImpersonatePrivilege        Impersonate a client after authentication Enabled
SeCreateGlobalPrivilege       Create global objects                     Enabled
SeIncreaseWorkingSetPrivilege Increase a process working set            Disabled
```

On remarque ici que `ContainerUser` ne dispose que de peu de droits et de privilèges sur le système, mais possède tout de même un jeton avec niveau d'intégrité élevé.

Volumes Les interactions des conteneurs avec des systèmes de fichiers extérieurs se font à deux niveaux : soit par l'image et ses couches successives lors de la création ou de l'instanciation d'un conteneur, soit par les volumes qui peuvent être montés dans le conteneur pour autoriser la persistence des données.

Les volumes sont implémentés sous Linux par des *bind mounts*, et sous Windows par des *NTFS Reparse points*[24] et plus précisément par des liens symboliques. Ceux-ci fonctionnent de manière similaire à ceux que l'on rencontre sous Linux et mènent à la sous-arborescence `\ContainerMappedDirectories\` de l'*object namespace* :

```
PS C:\> docker run -it -v C:\Users\:C:\shared microsoft/nanoserver powershell.exe
[CONTAINER] PS C:\> Get-Item C:\\shared | Select -Property LinkType,Target,Attributes

LinkType     Target                                          Attributes
--------     ------                                          ----------
SymbolicLink {\ContainerMappedDirectories\2C2A2195-[...]} Directory, ReparsePoint
```

Les volumes anonymes sont stockés dans `/var/lib/docker/volumes/` sous Linux et `C:\ProgramData\Docker\volumes\` sous Windows, aux côtés de nombreux fichiers de configuration. Il convient donc de vérifier les DACL des sous-arborescences `/var/lib/docker/` ou `C:\ProgramData\docker`. Sous Linux, tous les fichiers doivent appartenir à l'utilisateur `root` et aucun autre utilisateur ne doit avoir le droit de lire l'arborescence. Sous Windows, seule l'ACL héritée de `C:\ProgramData` protège par défaut les fichiers maintenus par Docker, ces derniers sont donc lisibles par tout le monde (incluant les volumes et fichiers de configuration contenant potentiellement des secrets), et tous les utilisateurs du système (`BUILTIN\Users`) peuvent y créer des fichiers ou y ajouter des données :

```
PS C:\> docker run -v C:\voltest -it microsoft/nanoserver powershell.exe
[CONTAINER] PS C:\> get-acl C:\voltest | fl

Path   : Microsoft.PowerShell.Core\FileSystem::C:\voltest
Owner  : NT AUTHORITY\SYSTEM
Group  : NT AUTHORITY\SYSTEM
Access : NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         NT AUTHORITY\SYSTEM Allow  FullControl
         CREATOR OWNER Allow  268435456
         BUILTIN\Users Allow  ReadAndExecute, Synchronize
         BUILTIN\Users Allow  AppendData
         BUILTIN\Users Allow  CreateFiles
    [...]
```

[24] https://msdn.microsoft.com/en-us/library/windows/desktop/aa365503(v=vs.85).aspx

Si un fichier est créé depuis le conteneur, celui-ci restera accessible en lecture pour tous les utilisateurs de l'hôte.

Pour éviter cela, on pourra désactiver l'héritage de l'ACL par défaut et ajouter une règle n'autorisant que le groupe `docker-users`, par exemple via les commandes :

```
PS C:\> icacls C:\ProgramData\Docker
C:\ProgramData\Docker NT AUTHORITY\SYSTEM:(I)(OI)(CI)(F)
                      BUILTIN\Administrators:(I)(OI)(CI)(F)
                      CREATOR OWNER:(I)(OI)(CI)(IO)(F)
                      BUILTIN\Users:(I)(OI)(CI)(RX)
                      BUILTIN\Users:(I)(CI)(WD,AD,WEA,WA)
PS C:\> icacls C:\ProgramData\Docker /inheritance:d
PS C:\> icacls C:\ProgramData\Docker /grant:r docker-users:F /remove:g *S-1-5-32-545
```

Élévation de privilèges au travers des volumes Deux points doivent être vérifiés lorsqu'on autorise l'accès à un stockage extérieur par un conteneur au travers d'un volume. Tout d'abord, les droits d'accès et en particulier l'utilisateur avec lequel le conteneur est exécuté qui va déterminer ce à quoi le conteneur pourra avoir accès, mais également l'étendue du volume que l'on exporte dans le conteneur.

Prenons le cas du groupe `docker-users` sous Windows (l'exemple marcherait aussi sous Linux avec le groupe `docker`) et ajoutons-y un utilisateur `NormalUser`, qui n'est pas membre du groupe local `Administrators` :

```
PS C:\> net user NormalUser | findstr 'Group'
Local Group Memberships      *docker-users         *Users
Global Group memberships     *None
```

La commande suivante va exécuter un conteneur en montant la racine du volume `C:\` de l'hôte dans le répertoire `C:\host\` à l'intérieur du conteneur :

```
PS C:\Users\NormalUser> docker run -it -v C:\:C:\host\ microsoft/windowsservercore
```

Une fois dans le conteneur, il est possible de créer un fichier dans le répertoire `C:\Windows\System32\` de l'hôte, car le jeton de sécurité du processus exécuté par le conteneur dispose du SID `S-1-5-32-544` (`BUILTIN\Administrators`) :

```
[CONTAINER] PS C:\> New-Item -Path C:\\host\\windows\\system32\\hostfile | Get-Acl | fl

Path   : Microsoft.PowerShell.Core\FileSystem::C:\host\windows\system32\hostfile
Owner  : User Manager\ContainerAdministrator
Group  : User Manager\ContainerAdministrator
Access : NT AUTHORITY\SYSTEM Allow  FullControl
         BUILTIN\Administrators Allow  FullControl
         BUILTIN\Users Allow  ReadAndExecute, Synchronize
         User Manager\ContainerAdministrator Allow  FullControl
   [...]
```

Lorsqu'un dossier de l'hôte est monté en écriture dans un conteneur, ce dernier peut potentiellement y écrire en tant que `ContainerUser` (`S-1-5-93-2-2`, SID inconnu à l'extérieur du conteneur). La liste de contrôle d'accès (ACL) NTFS la plus courante héritée de `C:\` lui permet de créer de nouveaux fichiers et dossiers. S'il est `ContainerAdministrator` (cas par défaut, mais à éviter tant que possible), il est membre du groupe `BUILTIN\Administrators` et ce dernier a un large accès en écriture au système de fichiers via la même ACL.

```
PS C:\Users\Administrator> Enter-PSSession -ContainerID 308778aa3a67[...]

[308778aa3a67...]: PS C:\Users\ContainerUser\Documents> whoami
user manager\containeruser

[308778aa3a67...]: PS C:\Users\ContainerUser\Documents> New-Item C:\\containerfile
New-Item : Access to the path 'C:\containerfile' is denied.
  + CategoryInfo          : PermissionDenied: (C:\containerfile:String) [New-Item],
                            UnauthorizedAccessException
  + FullyQualifiedErrorId : NewItemUnauthorizedAccessError,Microsoft.PowerShell.
                            Commands.NewItemCommand

[308778aa3a67...]: PS C:\Users\ContainerUser\Documents> New-Item C:\\host\\hostfile
New-Item : The file 'C:\host\hostfile' already exists.
  + CategoryInfo          : WriteError: (C:\host\hostfile:String) [New-Item], IOException
  + FullyQualifiedErrorId : NewItemIOError,Microsoft.PowerShell.Commands.NewItemCommand
```

Pour empêcher l'écriture arbitraire il est possible de préciser qu'un montage de volume doit être fait en lecture seule, par le modificateur `RO` de l'option `--volume C:\:C:\host\:RO`.

Aussi, il convient de n'exporter dans un conteneur que les répertoires strictement nécessaires à son fonctionnement, par exemple pour le stockage des données applicatives ou d'éléments de journalisation.

Isolation des volumes Pour les WSC, l'isolation des systèmes de fichiers repose sur l'isolation de l'*object namespace* (comme vu avec l'exemple des lettres de lecteurs en 6.2). Cette isolation empêche d'obtenir une *handle* vers un périphérique, volume, ou fichier d'un volume qui n'est pas exposé au WSC. A contrario, les Hyper-V containers reposent sur l'isolation fournie par Hyper-V et le partage de fichiers restreint implémenté sur *VMBus*. Il s'avère que dans le cas des WSC, l'exposition d'un volume partagé par un *reparse point NTFS* donne la possibilité de répliquer la vulnérabilité *"shocker"* découverte en juin 2014[25] dans des conteneurs Linux comme Docker ou les LXC privilégiés[26].

[25] `https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/`

[26] `https://lists.linuxcontainers.org/pipermail/lxc-users/2014-June/007248.html`

Cette vulnérabilité venait de l'appel système `open_by_handle_at()`, qui permet (s'il n'est pas bloqué et que l'on possède la *capability* `CAP_DAC_READ_SEARCH`) d'ouvrir, lire, et modifier des fichiers quelconques de l'hôte à partir d'un seul fichier partagé dans un conteneur. En effet, cette fonction donne un deuxième descripteur de fichier, étant donné un premier vers un fichier d'un système de fichiers, un numéro d'*I-node* (on utilisera 2, la racine du volume) et un numéro de version (facilement deviné).

Windows possède un appel système semblable, `OpenFileById()`, qui permet de même d'accéder à tout le système de fichiers à partir d'un dossier partagé dans les `\ContainerMappedDirectories\` d'un WSC. On peut par exemple utiliser le *FileId* 0x0005000000000005 pour obtenir une *handle* vers la racine du volume NTFS, puis parcourir toute l'arborescence grâce à cette handle passée à `NtCreateFile()` dans `OBJECT_ATTRIBUTES.RootDirectory`. Ce défaut d'isolation a été corrigé dans Windows Server 1803, mais demeure pour l'instant dans Windows Server 1709 et 2016. On préfèrera donc pour cette raison supplémentaire les Hyper-V containers pour tous les cas d'usage nécessitant une frontière de sécurité.

7.5 Registres, images et Dockerfiles

Registres Docker utilise le Docker Hub (*docker.io*), registre public fourni par Docker Inc. et proposant gratuitement de rendre n'importe quelle image accessible publiquement. D'autres registres peuvent être ajoutés, tels que *registry.fedoraproject.org* (ajouté automatiquement dans le paquet Docker distribué par Fedora). Tout le monde peut également héberger son propre registre au travers de l'image officielle `registry`, mais par défaut aucune authentification n'est présente et HTTPS n'est pas activé :

```
# docker pull registry
Status: Downloaded newer image for registry:latest
# docker run -d -p 5000:5000 --restart=always --name registry registry
# docker push localhost:5000/my_new_image
```

Si l'information qu'une image ou version particulière est utilisée est considérée comme confidentielle dans le contexte d'un audit, on vérifiera que les registres sont contactés en HTTPS. L'intégrité des images est protégée par un autre mécanisme décrit ci-après.

Images Docker sépare l'utilisation des images en deux étapes :

1. la création d'une image, sous forme d'un système de fichiers statique, composé de couches `aufs` ou `overlayfs` par exemple ;
2. l'instanciation des images sous forme de processus utilisant l'image comme système de fichiers racine.

Docker a été principalement conçu pour exécuter des conteneurs applicatifs minimalistes et immuables (« en lecture seule »), par opposition aux conteneurs système proposés plutôt par LXC et LXD qui ont, quant à eux, un cycle de vie proche de celui d'une machine virtuelle.

Les images Docker ne sont pas seulement de simples arborescences de systèmes de fichiers issues d'une distribution Linux (par l'intermédiaire de `debootstrap` par exemple) mais elles peuvent être créées à partir de n'importe quelle base au travers des fichiers Dockerfile. Le contenu de ces derniers est une suite de commandes décrivant le contexte du conteneur (ports réseau en écoute, partages de fichiers avec l'hôte, etc.) mais aussi exécutant des commandes à l'intérieur du conteneur, par exemple pour installer des paquets ou compiler un programme.

```
FROM debian:stretch-slim

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

RUN ln -sf /dev/stdout /var/log/nginx/access.log \
  && ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80

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

L'image est ensuite générée avec la commande `docker build`. Une fois l'image générée, elle peut être instanciée au travers de la commande `docker run`, ou en appelant l'API REST exposée par le serveur `containerd`.

Il est également possible de publier des images dans un registre, avec une syntaxe proche de celle de `git` (commit, push, pull).

À l'heure actuelle, peu d'images Docker sont disponibles pour Windows[27] et les deux plus utilisées comme images de base sont :

- microsoft/nanoserver (environ 130 Mo, contient le *framework* .NET 4.5 et un système Windows minimal) ;
- microsoft/windowsservercore (environ 3 Go, contient un système Windows quasi-complet).

Il n'existe pas non plus de systèmes de fichiers tels que *overlayfs* ou *aufs*, intégrant la notion de couches, comme sous Linux. Docker implémente

[27] `https://hub.docker.com/u/microsoft/`

donc un *graphdriver* spécifique à Windows, appelé `windowsfilter`, afin d'émuler ce système de couches au travers de liens symboliques. Les différentes couches des images sont stockées par défaut dans le répertoire `C:\ProgramData\Docker\windowsfilter` par identifiant d'image et de conteneur.

```
PS C:\> docker image inspect --format='{{.GraphDriver.Data.dir}}' microsoft/nanoserver
C:\ProgramData\Docker\windowsfilter\f95aa21ce8e0476911b0fdc7e05f68aed00679b5277d6[...]

PS C:\> docker container inspect --format='{{.GraphDriver.Data.dir}}' CONTAINERID
C:\ProgramData\Docker\windowsfilter\0c1218c5b2c5d80c7bc6b9b98ca482ac0bdda43c7e7bf[...]
```

L'image d'un conteneur Windows contient plusieurs fichiers et sous-répertoires :

```
PS C:\> get-childitem C:\ProgramData\Docker\windowsfilter\f95aa21ce8e0476[...]

    Directory: C:\ProgramData\Docker\windowsfilter\f95aa21ce8e0476[...]

Mode        LastWriteTime  Length Name
----        -------------  ------ ----
d-----  12-Feb-18   16:59        Files           # fichiers de l'image
d-----  12-Feb-18   17:09        Hives           # registre de l'image
d-----  09-Mar-18   16:46        UtilityVM       # machine virtuelle utilisée
                                                 # en mode Hyper-V
-a----  09-Mar-18   16:46   16384 bcd.bak         # environnement UEFI
-a----  09-Mar-18   16:46   12288 bcd.log.bak
-a----  09-Mar-18   16:46       0 bcd.log1.bak
-a----  09-Mar-18   16:46       0 bcd.log2.bak
-a----  09-Mar-18   16:46     108 layerchain.json # fichier contenant la liste
                                                 # des couches inférieures
```

L'abstraction créée par `docker pull` ne doit pas faire oublier les pratiques élémentaires de vérification d'intégrité et d'authenticité lorsque l'on télécharge de (potentiellement nombreux) exécutables. Il convient d'utiliser des registres officiels comme le Docker Hub, ainsi que des images maintenues à jour par des développeurs de confiance (Docker s'efforce de fournir de telles images dans la partie Explore du Docker Hub). Docker intègre pour cela Notary, un client et serveur basé sur *The Update Framework*[28] qui permet de servir n'importe quels contenus à des clients de sorte qu'ils puissent en vérifier l'intégrité, l'authenticité et l'état de mise à jour. Par défaut aucune vérification de signature n'est opérée, mais la variable d'environnement `DOCKER_CONTENT_TRUST` permet de restreindre les images et tags accessibles à ceux qui sont signés. Les images fournies par Microsoft ne sont malheureusement pas signées à ce jour. On montre ici l'échec d'un `docker pull` d'une image non signée avec *Docker Content Trust* (DCT) activé :

[28] `https://theupdateframework.github.io/`

```
$ notary -s https://notary.docker.io list docker.io/library/postgres
NAME                DIGEST                      SIZE (BYTES)    ROLE
----                ------                      ------------    ----
10                  e2688f79c920bbd5bcb8...     2371            targets
10-alpine           89c84fcccc147d403916...     2035            targets
[...]
$ notary -s https://notary.docker.io list docker.io/circleci/postgres
fatal: notary.docker.io does not have trust data for docker.io/circleci/postgres

$ export DOCKER_CONTENT_TRUST=1
$ docker pull circleci/postgres           # Pull impossible avec DCT
Error: remote trust data does not exist for docker.io/circleci/postgres [...]
$ docker pull library/postgres           # Mais possible pour une image signée
Digest: sha256:e2688f79c920bbd5bcb8e1ed54aef522c7e93a1a5eab32e10b4b020d49b4b925
Status: Downloaded newer image for docker.io/postgres@sha256:e2688f79c920bbd...
```

Dans le cas où des images sont poussées vers un registre privé, l'audit devra aussi porter sur l'administration du serveur en question. Si les images sont signées localement par des développeurs, l'audit devra inclure la gestion des clés Notary. Plusieurs types de clés sont impliqués :

- *root key* : elle signe toutes les autres clés et détermine pour chaque catégorie le quorum minimal de telles clés à atteindre (une par défaut pour DCT). Sa partie publique est envoyée au registre lors du premier push authentifié. La partie privée est stockée par défaut dans `/.docker/trust/private/root_keys`, chiffrée en AES-CBC 256 bits avec une clé dérivée d'un mot de passe par PBKDF2 (2048 itérations de SHA-1). Sa compromission entraîne la perte d'intégrité du dépôt ;
- *snapshot key* : elle signe tous les fichiers de métadonnées, afin qu'un attaquant ne puisse pas présenter une version d'une partie des fichiers et une autre version du reste. Cette clé est stockée sur le serveur, sa compromission permet le rejeu d'anciennes versions de fichiers ;
- *timestamp key* : elle est stockée sur le serveur pour périodiquement contre-signer la signature de la *snapshot key*, sa compromission permet le rejeu d'anciennes versions du dépôt ;
- *mirror key* : elle signe optionellement une liste de serveurs mirroirs depuis lesquels les mêmes opérations peuvent être réalisées, sa compromission n'est pas critique ;
- *target key* : elle signe la liste des condensats cryptographiques des fichiers du dépôt, ainsi que les *delegation keys* autorisées pour chaque sous-arborescence. Elle est stockée par défaut dans `/.docker/trust/tuf_keys/<nom du dépôt>/`, chiffrée comme la *root key*, mais avec un mot de passe différent. Sa compromission entraîne la perte d'intégrité du dépôt ;

– *delegation keys* : elles remplissent la même fonction que la *target key* pour tout ou partie du dépôt, mais peuvent être révoquées ou ajoutées en ne connaissant que la *target key*.

Les permissions appliquées aux fichiers contenant ces clés ainsi que la robustesse de leurs éventuels mots de passe doivent être vérifiées. La *root key* est particulièrement importante et n'est pas utilisée pour les mises à jour courantes, elle devrait donc être déplacée vers un stockage hors-ligne. L'idéal est un support matériel dédié, comme les Yubikey (prises en charge depuis Docker 1.11).

Seuls les identifiants des clés autorisées par délégation peuvent être récupérés, on devra donc les lister et trouver un moyen de vérifier leur légitimité :

```
$ notary -s https://notary.docker.io delegation list docker.io/testx/delegation
ROLE             PATHS           KEY IDS                         THRESHOLD
----             -----           -------                         ---------
targets/releases "" <all paths>  90256dc9d39b81482366fc6...    1
```

Chaque image doit être créée à partir de l'image de base la plus minimaliste possible, afin de limiter sa surface d'attaque et le nombre de mises à jour de sécurité qu'il faudra lui appliquer. La distribution *Alpine Linux* est par exemple très légère et bien maintenue.

Il est bien entendu possible de mettre à jour les conteneurs dynamiquement, en utilisant les ressources du système d'exploitation exécuté à l'intérieur (`yum` ou `apt-get` sous Linux par exemple). Microsoft fournit un outil appelé `cupdate.exe` (ou `ContainerUpdater.exe`) qui remplit également cette tâche :

```
[CONTAINER] PS C:\Windows\System32> cupdate.exe
[...]
Downloading updates...
[...]
Update title: 2018-03 Cumulative Update for Windows Server 2016
              for x64-based Systems (KB4088889)
Update contains 1 bundled updates.
        Bundled update:         Windows10.0-KB4088889-x64.cab
[...]
Total of 2 updates downloaded.
```

Cependant, les images instanciées puis retransformées en image (via la commande `docker commit`) sont à proscrire, étant donné leur manque de tracabilité et de reproductibilité. De telles images rencontrées en audit ne peuvent être analysées que partiellement : contrairement à un Dockerfile, la commande `docker history` indiquera quelle image est utilisée comme base, mais pas toutes les commandes qui y ont été exécutées.

La mise à jour de conteneurs applicatifs passe par la mise à jour de leur image. Ainsi, le processus est ici :

- re-création de l'image avec des composants à jour ;
- remplacement des anciens conteneurs par de nouvelles instances avec une image à jour.

Un conteneur applicatif devant être immuable, les données qu'il manipule (données applicatives ou journaux par exemple), doivent être exportées sur un espace de stockage en dehors du conteneur. Dans le cas des conteneurs système, la mise à jour aurait été réalisée par l'intermédiaire du gestionnaire de paquets de la distribution installée dans le conteneur.

Dockerfiles Au même titre que les images de base, les Dockerfiles doivent être contrôlés avant d'être utilisés. Pour que les images soient reproductibles, leurs Dockerfiles seront souvent partagés, par conséquent il ne doivent pas contenir de secrets ou ajouter de fichier en contenant à l'image. Si l'image ne contient qu'un petit nombre de fichiers binaires, il est possible d'utiliser une image vide (`FROM scratch`). Dans le cas opposé d'images devant contenir un environnement de compilation, on privilégiera un *multi-stage build* (supporté depuis Docker 17.05) :

```
FROM large/image AS build-env       # Premier environnement lourd
COPY src/ .                         # avec des outils de compilation
RUN make
FROM slim/other-image               # Deuxième environnement léger
COPY --from=build-env bin/myapp .   # avec le strict nécessaire pour l'exécution
```

On prêtera attention à l'utilisation de l'instruction `ADD`, qui peut télécharger des fichiers distants si son premier paramètre est une URL (on lui préfèrera l'instruction `COPY`), et plus généralement au téléchargement depuis des sources non contrôlées (exécutables non signés, dépôts ajoutés au gestionnaire de paquet de l'image sans vérification de signature, etc.) Pour finir, pour journaliser les événements indicateurs d'une compromission, il est important de respecter la limite d'un exécutable par conteneur, celui-ci devant être lancé par Docker (via l'instruction `CMD` ou `ENTRYPOINT`) et configuré pour écrire ses journaux d'activité sur sa sortie standard. Faute de support de l'applicatif, un serveur de collecte centralisé ou un volume partagé permettant d'écrire les journaux sur disque peut être utilisé.

8 Pistes d'amélioration

8.1 Fonctionnalités dérivées des *server silos*

Certaines nouvelles fonctionnalités de Windows 10 réutilisent les mêmes mécanismes bas niveau que les *Windows Server Containers* : il s'agit des *application silos* et de *Windows Defender Application Guard* (WDAG).

Les application silos ne tirent partie que de la virtualisation du système de fichiers et du registre d'un silo, mais gardent accès à tout leur hôte : le but n'est que d'augmenter la portabilité des applications, pas directement leur sécurité.

WDAG consiste quant à lui à instancier Microsoft Edge ou Internet Explorer dans un conteneur Hyper-V dédié. Dans son utilisation basique, l'utilisateur sélectionne les liens à ouvrir spécifiquement dans WDAG (sites « dangereux » par exemple). Dans un domaine, WDAG est configuré par GPO pour n'ouvrir que les liens de confiance hors de son conteneur (et jamais dans son conteneur). Tous les paramètres de ce mécanisme (liste d'URLs, préservation des fichiers téléchargés, cookies et autres modifications du système de fichier, etc.) méritent d'être vérifiés lors d'un audit.

8.2 Protection des conteneurs vis-à-vis de l'hôte

Un attaquant qui aurait déjà pris la main sur l'hôte et y disposerait de privilèges suffisants pour interagir avec les conteneurs (souvent `root` ou membre d'un groupe privilégié comme le groupe `docker`) pourrait mettre en cause l'intégrité des conteneurs ainsi que la confidentialité des données qui y seraient manipulées. Ceci peut également être le cas de conteneurs exécutés dans un *cloud* public.

Des fonctionnalités des processeurs récents, telles que les *Secure Enclaves* Intel SGX ou AMD SEV, pourraient apporter un début de réponse à cette problématique [6], en protégeant la mémoire des applications exécutées dans les conteneurs. Cependant, le mécanisme de *Secure Enclave* laisse à l'application la tâche de chiffrer correctement ses informations sensibles avant de les faire sortir de l'*enclave*. De plus, l'application doit vérifier exhaustivement les informations données par l'OS avant de les utiliser, ce qui impose de l'adapter ou la recoder en repensant complètement son modèle de sécurité [12]. Le projet SCONE [11] vise par exemple à fournir une protection générique contre ces attaques, pour protéger n'importe quel conteneur. Cependant, aucun projet ne répond aux attentes en terme de protection contre les attaques par canaux auxiliaires [13], ou vis-à-vis

des décisions controversées d'Intel affaiblissant le mécanisme d'attestation d'enclave à distance.

8.3 Filtrage inter-conteneurs

Par ailleurs, des logiciels comme Cilium[29] exploitent de nouvelles pistes. Ce dernier se présente comme une couche intermédiaire utilisant les fonctionnalités des filtres eBPF fournies par le noyau Linux et propose de remplacer les pare-feu réseau pour les communications inter-conteneurs.

9 Conclusion

Docker est aujourd'hui un acteur principal et incontournable et il tend à s'intégrer dans de plus en plus d'environnements, comme en témoignent les évolutions majeures apportées aux systèmes Microsoft. Il est donc primordial d'en connaître les forces et les faiblesses et d'avoir la possibilité d'en évaluer la sécurité de la manière la plus automatisable possible.

Cependant, l'écosystème des technologies de conteneurs est très volatil et il n'est pas rare de voir apparaître ou disparaître des protagonistes quasiment du jour au lendemain. Ainsi, le savoir doit porter autant sur les mécanismes bas niveau, communs à de nombreuses solutions, qu'aux solutions elles-mêmes. Il faut s'assurer de la sécurité de bout en bout, de l'étape de développement des applications hébergées dans les conteneurs à l'orchestration et au déploiement de ces dernières, en passant par la création des images et l'isolation de chaque brique.

Références

1. Aaron Grattafiori. Understanding and Hardening Linux Containers. `https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/`, 2016.
2. Jesse Hertz. Abusing Privileged and Unprivileged Linux Containers. `https://www.nccgroup.trust/us/our-research/abusing-privileged-and-unprivileged-linux-containers/`, 2016.
3. Adrien Mouat. Docker Security. `https://www.safaribooksonline.com/library/view/docker-security/9781492042297/`, 2015.
4. Alex Ionescu. Helium, Argon, Krypton & Xenon : The Noble Gases of Windows Containers. `http://www.alex-ionescu.com/publications/syscan/syscan2017.pdf`, 2017.
5. Chris Foster. Privilege Escalation via Docker. `https://fosterelli.co/privilege-escalation-via-docker.html`, 2015.

[29] `https://github.com/cilium/cilium`

6. Jonathan Corbet. Two approaches to x86 memory encryption. https://lwn.net/Articles/686808/, 2016.
7. Janak Desai. new system call, unshare. https://lwn.net/Articles/135266/, 2005.
8. Michael Kerrisk. Anatomy of a user namespaces vulnerability. https://lwn.net/Articles/543273/, 2005.
9. Michael Kerrisk. Namespaces in operation, part 1 : namespaces overview. https://lwn.net/Articles/531114/, 2013.
10. Eric W. Biedeman. Multiple Instances of the Global Linux Namespaces. https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf, 2006.
11. Sergei Arnautov. SCONE : Secure Linux Containers with Intel SGX. https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf, 2016.
12. Stephen Checkoway. Iago Attacks : Why the System Call API is a Bad Untrusted RPC Interface. https://cseweb.ucsd.edu/~hovav/dist/iago.pdf, 2013.
13. Yuanzhong Xu. Controlled-Channel Attacks : Deterministic Side Channels for Untrusted Operating Systems. http://www.ieee-security.org/TC/SP2015/papers-archived/6949a640.pdf, 2015.