RAPPORT TP

Elèves:

Mathis FOUCADE
Jules PUGET
Axelle ROUZIER

Enseignant:

M LOUTSCH ETIENNE

Session 1: Introduction aux Containers et à la Sécurité

Paris 2025

Partie 1: Introduction aux containers

1. Lancer un Container Simple

```
axelle@securite-containeur:~$ docker run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
e6590344b1a5: Pull complete
Digest: sha256:7e1a4e2d11e2ac7a8c3f768d4166c2defeb09d2a750b010412b6ea13de1efb19
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Dans cette première étape, nous avons utilisé la commande *docker run --rm hello-world* pour lancer un container minimal. Ce test nous a permis de vérifier le bon fonctionnement de notre installation Docker. Le message affiché détaille le processus complet d'exécution d'un container: le client Docker contacte d'abord le daemon Docker, qui télécharge ensuite l'image depuis Docker Hub si elle n'est pas disponible localement. Le daemon crée alors un nouveau container à partir de cette image et exécute le programme qu'elle contient. Enfin, il transmet la sortie au client Docker qui l'affiche dans notre terminal. L'option *--rm* que nous avons utilisée permet de supprimer automatiquement le container après son exécution, ce qui évite d'accumuler des containers inutilisés sur notre système.

2. Explorer un Container en Interactif

```
axelle@securite-containeur:~$ docker run -it --rm alpine sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
f18232174bc9: Pull complete
Digest: sha256:a8560b36e8b8210634f77d9f7f9efd7ffa463e380b75e2e74aff4511df3ef88c
Status: Downloaded newer image for alpine:latest
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ # pwd
/
/ # whoami
root
/ #
```

Pour cette deuxième étape, nous avons exécuté la commande *docker run -it --rm alpine sh* qui nous a permis d'explorer un container en mode interactif. Alpine Linux est une distribution particulièrement légère, souvent utilisée comme base pour les containers. Nous avons utilisé les options *-i* pour maintenir l'entrée standard ouverte et *-t* pour allouer un pseudo-terminal, ce qui nous a permis d'interagir avec le shell du container. En utilisant des commandes comme *ls*, *pwd*, et *whoami*, nous avons pu constater que l'environnement est un système Linux minimal avec les dossiers standard. Un point important à noter est que l'utilisateur par défaut dans ce container est *root*, ce qui représente un risque de sécurité potentiel dans un environnement de production. Le container dispose de son propre système de fichiers isolé de l'hôte, ce qui est l'un des mécanismes d'isolation fournis par la technologie des containers.

3. Analyser les ressources système d'un container

```
axelle@securite-containeur:~$ docker run -d --name test-container nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
6e909acdb790: Pull complete
5eaa34f5b9c2: Pull complete
417c4bccf534: Pull complete
e7e0ca015e55: Pull complete
373fe654e984: Pull complete
97f5c0f51d43: Pull complete
c22eb46e871a: Pull complete
Digest: sha256:124b44bfc9ccd1f3cedf4b592d4d1e8bddb78b51ec2ed5056c52d3692baebc19
Status: Downloaded newer image for nginx:latest
066edda59710c2ed9d5c7b1563ce9ffadd127130f12acbb187a75ce098cd4a74
axelle@securite-containeur:~$ docker stats test-container

CONTAINER ID   NAME             CPU %     MEM USAGE / LIMIT     MEM %     NET I/
O       BLOCK I/O         PIDS
066edda59710   test-container   0.03%     4.449MiB / 1.922GiB   0.23%     2.96kB
 / 0B   1.65MB / 12.3kB   3
```

Pour cette troisième étape, nous avons lancé un serveur web nginx en arrière-plan avec la commande *docker run -d --name test-container nginx*, puis nous avons analysé ses ressources avec *docker stats test-container*. Les résultats montrent clairement l'un des avantages majeurs des containers: leur légèreté. Notre container nginx ne consommait qu'environ 0.03% de CPU et seulement 4.4MiB de mémoire sur les 1.92GiB disponibles. L'utilisation réseau était également minimale à 2.96kB. Ces chiffres confirment que les containers sont beaucoup plus économes en ressources que les machines virtuelles traditionnelles, car ils partagent le noyau du système d'exploitation hôte et n'utilisent que les ressources nécessaires à leur fonctionnement spécifique.

4. Lister les capacités d'un container

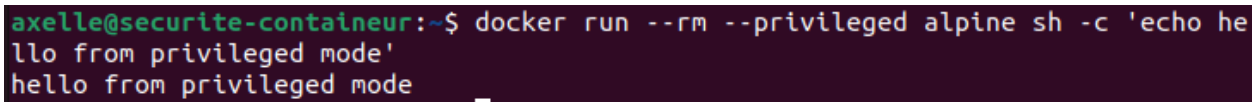
```
axelle@securite-containeur:~$ docker run --rm --cap-add=SYS_ADMIN alpine sh -c '
cat /proc/self/status'
Name:   cat
Umask:  0022
State:  R (running)
Tgid:   1
Ngid:   0
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 64
Groups: 0 1 2 3 4 6 10 11 20 26 27
NStgid: 1
NSpid:  1
NSpgid: 1
NSsid:  1
Kthread:        0
VmPeak:     1688 kB
VmSize:     1688 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       768 kB
VmRSS:       768 kB
RssAnon:               0 kB
RssFile:             768 kB
RssShmem:              0 kB
VmData:       88 kB
VmStk:       132 kB
VmExe:       616 kB
VmLib:       356 kB
VmPTE:        52 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
CoreDumping:    0
THP_enabled:    1
untag_mask:     0xffffffffffffffff
Threads:        1
SigQ:   2/7572
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000004
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 00000000a82425fb
CapEff: 00000000a82425fb
CapBnd: 00000000a82425fb
CapAmb: 0000000000000000
NoNewPrivs:     0
Seccomp:        2
Seccomp_filters:        1
Speculation_Store_Bypass:       vulnerable
SpeculationIndirectBranch:      always enabled
Cpus_allowed:   3
Cpus_allowed_list:      0-1
Mems_allowed:   00000000,00000000,00000000,00000000,00000000,00000000,00000000,0
0000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000
,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        13
nonvoluntary_ctxt_switches:     39
x86_Thread_features:
x86_Thread_features_locked:
```

Dans cette étape, nous avons exécuté un container avec des privilèges supplémentaires en utilisant la commande *docker run --rm --cap-add=SYS_ADMIN alpine sh -c 'cat /proc/self/status'*. L'option *--cap-add=SYS_ADMIN* accorde au container la capacité d'effectuer

des opérations administratives système, ce qui est normalement restreint pour des raisons de sécurité. Le fichier */proc/self/status* nous a montré les détails sur les capacités et les permissions du processus en cours d'exécution. Cette démonstration illustre comment Docker permet un contrôle fin des privilèges accordés aux containers. D'un point de vue sécurité, il est recommandé de suivre le principe du moindre privilège et de n'accorder que les capacités strictement nécessaires à l'application.

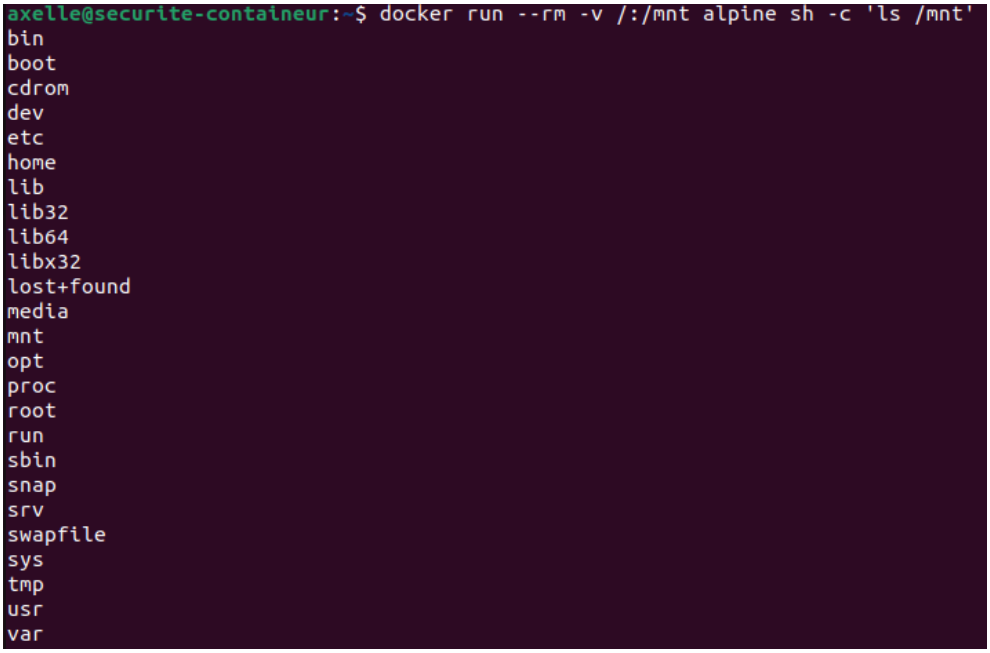
Partie 2: Vulnérabilités et menaces des containers

1. **Tester un Container avec des Permissions Élevées**



Nous avons lancé un container en mode privilégié avec la commande *docker run --rm --privileged alpine sh -c 'echo hello from privileged mode'*. L'option *--privileged* est particulièrement dangereuse car elle accorde au container pratiquement tous les privilèges du système hôte. Dans un environnement de production, cette configuration constitue un risque majeur car un attaquant qui compromettrait ce container pourrait potentiellement prendre le contrôle complet du système hôte. Cette démonstration met en évidence l'importance d'éviter les containers privilégiés dans les déploiements réels et de toujours appliquer le principe du moindre privilège.

2. **Simuler une Évasion de Container**



Pour cette étape, nous avons exécuté *docker run --rm -v /:/mnt alpine sh -c 'ls /mnt'* pour monter le système de fichiers complet de l'hôte dans le container. Cette configuration erronée simule ce qu'un attaquant pourrait faire lors d'une évasion de container. En montant le système de fichiers de l'hôte, nous avons pu voir tous les fichiers et dossiers de la machine hôte depuis le container. Cette faille de configuration est extrêmement dangereuse car elle brise l'isolation qui est au cœur de la sécurité des containers. Un attaquant pourrait lire des données sensibles, modifier des fichiers système ou même installer des logiciels malveillants sur l'hôte.

3. Créer une Image Sécurisée

Dans cette étape, nous avons créé un Dockerfile sécurisé avec le contenu suivant:

```
  GNU nano 6.2                          Dockerfile *
FROM alpine
RUN adduser -D appuser
USER appuser
CMD ["echo", "Container sécurisé!"]
```

```
axelle@securite-containeur:~/secure-container$ docker build -t sercure-image .
DEPRECATED: The legacy builder is deprecated and will be removed in a future rel
ease.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM alpine
 ---> aded1e1a5b37
Step 2/4 : RUN adduser -D appuser
 ---> Running in 83fdb983567f
 ---> Removed intermediate container 83fdb983567f
 ---> 00d49e7d1b83
Step 3/4 : USER appuser
 ---> Running in 78248d6678fb
 ---> Removed intermediate container 78248d6678fb
 ---> bacbd7007559
Step 4/4 : CMD ["echo", "Container sécurisé!"]
 ---> Running in b8dde0d6d6c3
 ---> Removed intermediate container b8dde0d6d6c3
 ---> 642eb080cf58
Successfully built 642eb080cf58
Successfully tagged sercure-image:latest
```

```
axelle@securite-containeur:~/secure-container$ docker run --rm sercure-image
Container sécurisé!
```

Après avoir construit l'image avec *docker build -t sercure-image .*, nous avons vérifié son fonctionnement avec *docker run --rm sercure-image*. Ce Dockerfile illustre une bonne pratique de sécurité fondamentale: l'utilisation d'un utilisateur non privilégié pour exécuter l'application dans le container. En créant l'utilisateur *appuser* et en spécifiant que les commandes doivent être exécutées sous cette identité via l'instruction *USER*, nous réduisons significativement les

risques liés à une éventuelle compromission de l'application. Même si un attaquant prend le contrôle de l'application, ses capacités seront limitées par les permissions restreintes de cet utilisateur.

4. **Restreindre l'accès réseau d'un container**
5. **Bloquer la connexion internet dans un container :**
6. **Tester l'accès internet avec par exemple `ping google.com`.**

Pour cette étape, nous avons d'abord lancé un container Alpine en arrière-plan avec la commande *docker run -d --name mon-container alpine sleep 3600*, créant ainsi un container qui reste actif pendant une heure. Ensuite, nous avons utilisé la commande *docker network disconnect bridge mon-container* pour déconnecter ce container du réseau bridge par défaut. Cette action est particulièrement importante d'un point de vue sécurité car elle isole complètement le container des communications réseau, y compris d'internet.

```
axelle@securite-containeur:~$ docker run -d --name mon-container alpine sleep 36
00
c5997ace0382a6f3d023f83102a9a3ec6a31218b2de3edfd5b83bf02a32bf072
axelle@securite-containeur:~$ docker network disconnect bridge mon-container
axelle@securite-containeur:~$ docker exec mon-container ping -c 3 google.com
ping: bad address 'google.com'
```

Après cette déconnexion, nous avons vérifié l'isolation en nous connectant au container avec *docker exec -it mon-container sh* et en tentant d'exécuter *ping google.com*. Comme prévu, cette commande a échoué avec l'erreur "bad address 'google.com'", confirmant que le container n'avait plus accès à internet. Cette technique d'isolation réseau est essentielle dans un environnement de production pour limiter le risque d'attaques latérales et appliquer le principe de défense en profondeur. Un container ne devrait avoir accès qu'aux ressources réseau strictement nécessaires à son fonctionnement. Cette approche réduit significativement la surface d'attaque et limite les possibilités d'un attaquant qui aurait réussi à compromettre un container.

7. **Télécharger et Scanner une Image**

```
axelle@securite-containeur:~$ docker pull vulnerables/web-dvwa
Using default tag: latest
latest: Pulling from vulnerables/web-dvwa
3e17c6eae66c: Pull complete
0c57df616dbf: Pull complete
eb05d18be401: Pull complete
e9968e5981d2: Pull complete
2cd72dba8257: Pull complete
6cff5f35147f: Pull complete
098cffd43466: Pull complete
b3d64a33242d: Pull complete
Digest: sha256:dae203fe11646a86937bf04db0079adef295f426da68a92b40e3b181f337daa7
Status: Downloaded newer image for vulnerables/web-dvwa:latest
docker.io/vulnerables/web-dvwa:latest
```

```
axelle@securite-containeur:~$ cat results.json | jq '.Results[] | .Vulnerabiliti
es | length'
1575
0
```

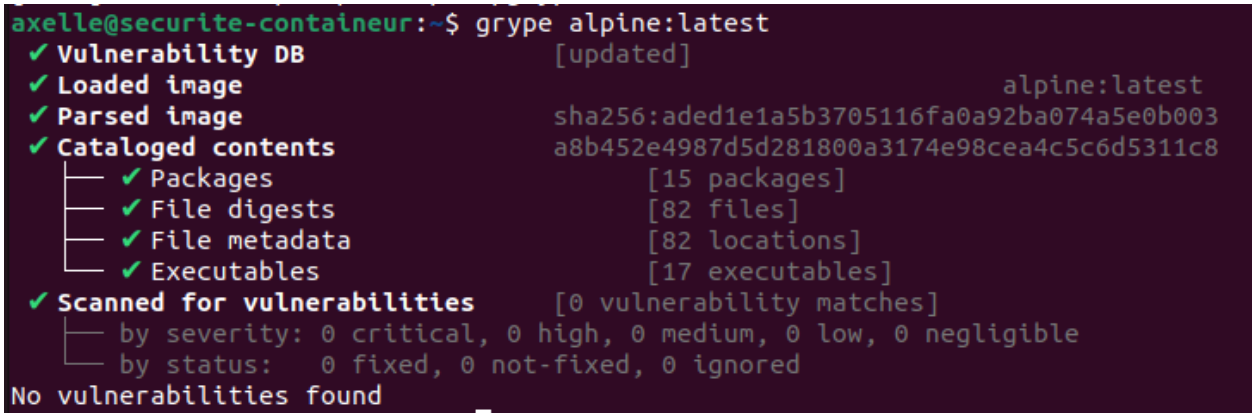
Dans cette étape, nous avons délibérément téléchargé une image connue pour ses vulnérabilités, *vulnerables/web-dvwa*, qui est une application web intentionnellement vulnérable utilisée pour l'apprentissage de la sécurité. Après avoir récupéré cette image avec *docker pull vulnerables/web-dvwa*, nous l'avons analysée en profondeur avec l'outil Trivy en utilisant la commande *trivy image vulnerables/web-dvwa*.

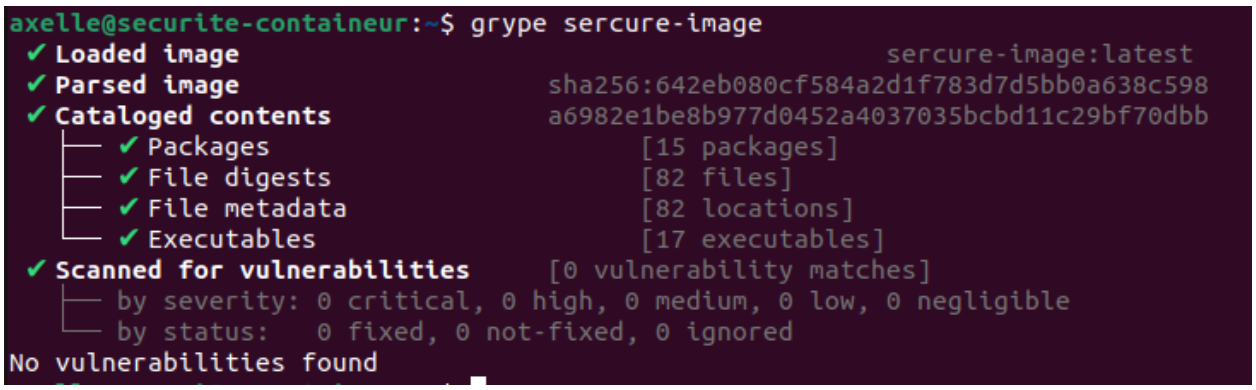
Les résultats du scan ont été particulièrement révélateurs, montrant un total impressionnant de 1575 vulnérabilités. Nous avons enregistré ces résultats détaillés dans un fichier JSON avec la commande *trivy image -f json -o results.json vulnerables/web-dvwa*, ce qui permet une analyse plus approfondie ou une intégration dans des outils de reporting automatisés.

En examinant le résumé des vulnérabilités, nous avons constaté que l'image contient de nombreuses failles critiques et à haut risque, principalement dans les bibliothèques PHP, les composants du système d'exploitation et les dépendances web. Parmi les vulnérabilités les plus préoccupantes, on trouve plusieurs injections SQL, des failles XSS (Cross-Site Scripting), des risques d'exécution de code à distance (RCE) et diverses vulnérabilités liées à des versions obsolètes de bibliothèques.

Cette analyse démontre l'importance cruciale de scanner systématiquement les images avant leur déploiement. Dans un environnement de production réel, il serait absolument impensable d'utiliser une image présentant un tel niveau de risque. Cette pratique souligne également pourquoi les entreprises devraient mettre en place des politiques strictes concernant les sources d'images autorisées, privilégier les images officielles ou maintenues en interne, et établir un processus rigoureux de vérification de sécurité dans le pipeline CI/CD.

8. **Scanner une Image pour Détecter les Vulnérabilités**





Pour cette dernière étape, nous avons utilisé deux outils différents pour analyser la sécurité de nos images:

1. Nous avons d'abord scanné l'image *alpine:latest* avec Grype, ce qui n'a révélé aucune vulnérabilité.
2. Nous avons ensuite comparé ce résultat avec notre image sécurisée personnalisée.
3. Nous avons également analysé l'image *vulnerables/web-dvwa* avec Trivy, qui a identifié 1575 vulnérabilités, démontrant l'importance de scanner régulièrement les images.

La différence entre Grype et Trivy réside principalement dans leur approche du scan. Trivy propose une analyse plus détaillée avec une catégorisation des vulnérabilités par sévérité, tandis que Grype offre une interface plus simple mais efficace. Les deux outils sont essentiels dans une stratégie de sécurité des containers, car ils permettent d'identifier les vulnérabilités avant le déploiement et de maintenir un environnement plus sûr. Cette étape souligne l'importance d'intégrer ces scanners dans le pipeline CI/CD pour assurer une sécurité continue des applications containerisées.