UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di
Ingegneria

Corso di laurea in
Ingegneria Informatica

Classe n. L-8

Vulnerabilità di Fuga da Docker

Candidato:
Giorgio Chirico

Matricola n.
1068142

Relatore:
Chiar.mo Stefano Paraboschi

Correlatore (se applicabile):
Dott. (Dott.ssa) Nome Cognome

Anno Accademico
2022/2023

ABSTRACT

I containers sono degli ambienti di lavoro e di runtime virtualizzati, divenuti popolari per la loro leggerezza, flessibilità e portabilità specie in contesto cloud. Una caratteristica importante è la presenza di uno strato di isolamento dal sistema operativo ospitante, basata sugli spazi d'indirizzi, che porta erroneamente a comparare questi ambienti a delle piccole macchine virtuali: proprio la mancata osservazione delle differenze tra una macchina virtuale e un container, oltre alle caratteristiche intrinseche dei containers o del framework scelto, può aprire la possibilità di una compromissione dell'isolamento tra il container e la macchina ospitante, portando a un'escalation dei privilegi, da parte degli utenti dei containers, sull'host.

Questo documento inizierà con un'introduzione delle caratteristiche del software Docker e della creazione dei container Docker, le impostazioni di default di un'installazione Docker e gli aspetti relativi alla sua sicurezza; seguirà quindi un'illustrazione variegata di contesti di vulnerabilità atti a dimostrare la possibilità di fuga sfruttando una o più proprietà dei Docker containers. Le vulnerabilità sfruttate possono essere frutto di una non attenta configurazione o di un problema 'storico' che caratterizza un certo intervallo di versioni del software Docker. In chiusura, verranno proposte delle forme di mitigazione 'ad ampio spettro' per aumentare la sicurezza dei containers.

L'illustrazione delle vulnerabilità ha lo scopo di aprire una riflessione su determinate caratteristiche del framework Docker. Le dimostrazioni sono presentate per mezzo di codici realizzati nei linguaggi Python3, C, Shell, Bash, Dockerfile. Ogni dimostrazione si pone come obiettivo l'ottenimento di una reverse-shell dall'host o di una bind-shell sull'host. A termine di ogni dimostrazione verranno spiegate le possibili soluzioni attuabili per una rapida mitigazione.

INDICE DEI CONTENUTI

pagina

1 INTRODUZIONE 1
1.1 Container 1
1.2 Spazi d'indirizzi 2
1.3 Docker 3
1.3.1 Generalità 3
1.3.2 Esempio: creazione ed esecuzione di un docker container 4
1.3.3 Esempio: docker create 4
1.3.4 Esempio: docker start 5
1.3.5 Docker container default namespaces 7
1.4 Utenti e gruppi 10
1.4.1 Utente del container e utente dell'host 10
1.4.2 Utente root in Docker container 10
1.5 Storage drivers in Docker 11
1.5.1 Gestione mounts 11
1.5.2 Volumi 11
1.5.3 OverlayFS (Overlay Filesystem) 11
1.5.4 OverlayFS in Docker container 12
1.6 Networking in Docker 15
1.7 Sicurezza predefinita in Docker 16
1.7.1 cgroups (control groups) 16
1.7.2 seccomp (Secure Computing) 16
1.7.3 AppArmor 17
1.7.4 SELinux 17
1.7.5 Capabilities e processi privilegiati 18
1.7.6 ACL delle componenti Docker 18
RIFERIMENTI 20

INDICE DELLE FIGURE

Pagina

Figura 1: creazione Docker container 5
Figura 2: inizializzazione Docker container 6
Figura 3: PID di una task eseguita da container..... 7
Figura 4: comando strace per tracciare il processo 'containerd'..... 7
Figura 5: ‘unshare’ syscall di containerd 8
Figura 6: confronto tra namespaces di task da container e bash da host..... 9
Figura 7: utente root in Docker container 10
Figura 8: funzionamento di OverlayFS..... 12
Figura 9: illustrazione di overlay2 14
Figura 10: ACL del Docker Unix socket (Docker API) 18

1 INTRODUZIONE

1.1 Container

Un container [1] è una tecnologia software che permette la veloce creazione, configurazione e attivazione di sistemi operativi o ambienti di sviluppo di applicativi sia Desktop che Cloud.

Punto di forza dei container è la loro leggerezza e indipendenza dalla piattaforma su cui vengono creati ed eseguiti: per mezzo di configurazioni pre-impacchettate, dette immagini [2], è possibile la costruzione di questi container, la cui esecuzione può esser configurata a seconda delle esigenze.

Esistono diversi framework per la gestione dei container, attraverso riga di comando o gestore applicativo. Generalmente, questi framework si distinguono in:

- daemon-based: un controllore centrale, detto 'demone', gestisce le richieste d'interazione con l'ambiente, i container e le immagini: un esempio popolare è Docker.
- daemonless: senza demone, ad esempio Podman.
- root: i componenti del framework, come ad esempio il demone, appartengono all'utente 'root' (amministratore) .
- rootless: i componenti del framework non appartengono a 'root', divenendo eseguibili in maniera non privilegiata.

I container forniscono un minimo grado di isolamento, rispetto all'host, dei processi attivi al loro interno per mezzo dei namespace, ovvero gli 'spazi di indirizzi': i processi interni al container creato 'non hanno idea' di essere parte di un sottospazio dell'host.

Va rimarcato che l'isolamento provvisto dai containers è solo fittizio: mentre lo spazio utente viene interamente astratto, lo spazio kernel è lo stesso dell'host. Questa è un'importante differenza rispetto alle macchine virtuali, oltre al fatto che quest'ultime provocano un maggiore overhead in fase d'esecuzione.

1.2 Spazi d’indirizzi

Gli spazi d’indirizzi, anche detti ‘namespaces’, sono una caratteristica dei kernel Linux [3]. Servono a isolare le risorse di un certo gruppo di processi dalle risorse di altri gruppi di processi: nel caso dei containers, permettono l’isolamento dei processi interni al container rispetto ai processi esterni al container.

Ogni spazio d’indirizzo virtualizza una certa risorsa:

- cgroup namespace: gestore dei cgroups, ovvero i gruppi di controllo per la gestione delle risorse logiche e fisiche da parte dei processi interni, i quali vengono classificati in gruppi;
- user namespace: gestore di User ID, Group ID e capabilities all’interno del container. Inoltre, fa sì che i privilegi utente interni al namespace non siano gli stessi sull’host;
- mnt namespace: gestore dei punti di mount interni al container;
- net namespace: gestore delle risorse network interne al container;
- uts namespace: gestore dell’hostname del container;
- ipc namespace: gestisce le risorse di Inter Process Communication, come le message queues, interne al container;
- pid namespace: gestore degli identificativi di processo (PID) all’interno del container.

1.3 Docker

1.3.1 Generalità

Docker è un framework open-source, sviluppato dalla Docker Inc. , per lo sviluppo e l'impiego di container Linux, cioè containers basati su kernel Linux [4] .

Docker è daemon-based e, di base, root. Esso è sviluppato come un'architettura client-server conforme alle regole REST API. Esso è disponibile anche come applicativo desktop.

Di base, la sua architettura è così costituita:

- lato client, il Docker CLI: dotato interfaccia a riga di comando o GUI per interagire col framework Docker per mezzo di richieste HTTP;
- lato server, Docker Engine: una RESTful API che risponde alle richieste della Docker CLI e gestisce il sistema Docker;
- sul cloud, le registry: repositories di immagini a cui il framework, anche automaticamente, fa richiesta per ricevere le immagini mancanti in locale, o su cui è possibile salvare, con un proprio account, le proprie immagini.

Riguardo al Docker Engine, risultano fondamentali le componenti che seguono:

- *dockerd*: anche detto Docker Daemon, resta in ascolto di default su un UNIX socket, in attesa di richieste conformi all'API di Docker. Ha ruolo di controllore centrale per l'intero sistema Docker: gestisce gli oggetti Docker quali immagini,containers,volumi,networks e le task di alto livello quali, ad esempio, login,build,inspect, pull. Può esser posto in ascolto su un TCP socket;
- *containerd* [5]: il Container Daemon, gestisce la container runtime. La maggior parte delle interazioni a basso livello sono gestite da una componente al suo interno chiamata runc [6];
- *runc* [7]: una runtime indipendente che garantisce la portabilità dei containers conformi agli standard. Tra le sue caratteristiche, spicca il supporto nativo per tutti i componenti di sicurezza Linux come, ad esempio, Apparmor, seccomp, control

groups, capabilities. Ha completo supporto dei Linux namespaces, inclusi user namespaces: è responsabile della creazione dei namespace ed esecuzione dei containers.

In particolare, runc è invocata da containerd-shim [8]: processo figlio di containerd e parente diretto del container che verrà creato. Tale processo è responsabile dell'intero ciclo di vita del container e delle logiche di riconnessione. Anche la gestione dei containerd-shim avviene tramite API [9] .

1.3.2 Esempio: creazione ed esecuzione di un docker container

Si ponga in ipotesi di voler creare ed eseguire un container di nome C1 utilizzando un'immagine 'alpine' (sistema operativo esclusivo per immagini leggere in Docker) della versione più recente ('latest'). Si possono impartire entrambe le operazioni per mezzo del comando 'docker run' , ma per un'illustrazione semplificata di come funziona l'architettura Docker è possibile scomporre tale comando nelle sue operazioni costituenti, per mezzo dei due comandi unitari che esso va a sostituire:

1. docker create
2. docker start

Le risposte di dockerd dipendono dalla richiesta API , secondo gli standard della API documentation di riferimento [10].

1.3.3 Esempio: docker create

1. Il comando di creazione, impartito dalla Docker CLI, viene convertito in richiesta HTTP e inviata in modalità POST.
2. dockerd verifica la correttezza della richiesta: qualora non fosse correttamente formata, invierà una risposta d'errore.
3. dockerd verifica la presenza dell'immagine nella repository locale: qualora fosse mancante, invierà una richiesta di pull al registry (ad esempio, Docker Hub) per

scaricare l'immagine. Se tale immagine risultasse mancante anche sulla registry, dockerd risponderà al client con un errore.

4. dockerd crea il nuovo container partendo dall'immagine specificata, ma non lo inizializza: seguendo il meccanismo Copy-On-Write, viene quindi creato un layer scrivibile sopra ai layer dell'immagine [11, 12].
5. Finita la corretta esecuzione, dockerd invia una risposta positiva alla docker CLI.

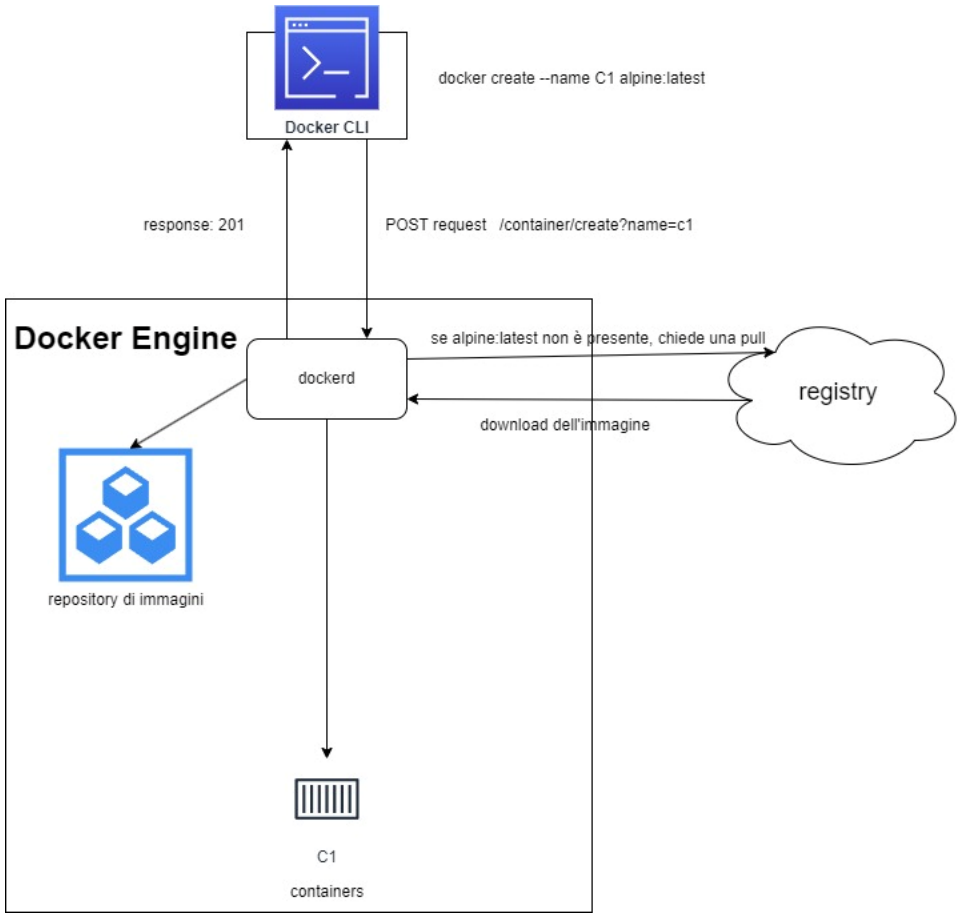

Figura 1: creazione Docker container

1.3.4 Esempio: docker start

1. il comando di inizializzazione impartito dalla Docker CLI viene convertito in richiesta HTTP e inviata in modalità POST.
2. dockerd verifica la correttezza della richiesta: qualora non fosse correttamente formata, invia risposta d'errore.

3. dockerd verifica l'esistenza del container di nome 'C1': qualora assente, risponde con un errore.
4. dockerd chiede a containerd l'esecuzione del container 'C1'.
5. containerd crea un processo containerd-shim per la gestione del ciclo di vita del container 'C1': quest'ultimo processo invoca runC.
6. runC completerà la configurazione di runtime del container. In particolare, eseguirà la syscall '*unshare*' per la creazione dei namespaces interni al container: Successivamente, eseguirà una fork del processo di init (PID 1) all'interno del container.
7. Una volta che l'esecuzione del container è inizializzata, runC termina.
8. Finita la corretta esecuzione, dockerd invia risposta positiva alla docker CLI.

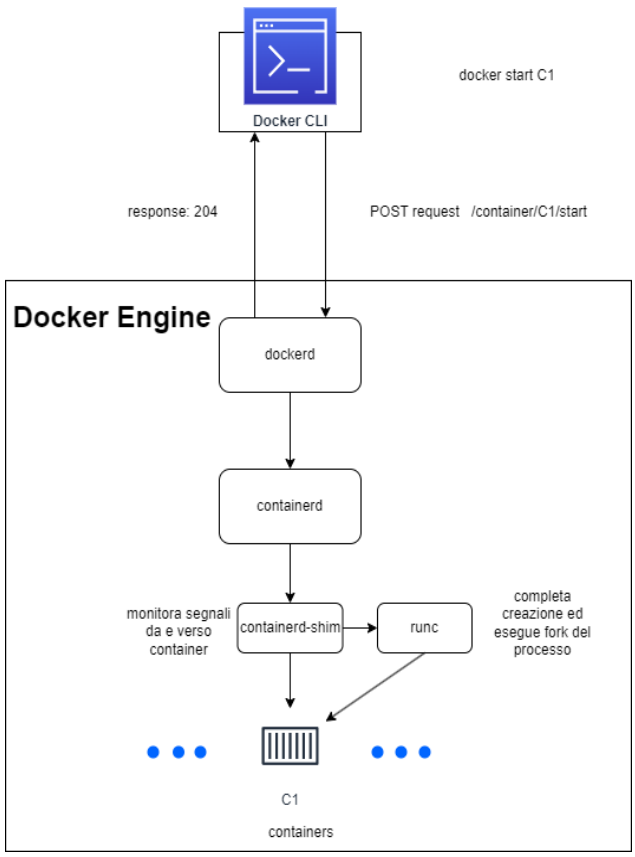

Figura 2: inizializzazione Docker container

Va rimarcata la natura fittizia di questo isolamento: il processo con PID 1 nel container non è il processo init dell'host. Il suo 'vero' PID potrebbe essere, per esempio, 5753: di conseguenza, tutti i processi che verranno creati da init avranno PID 2,3,4 … nel container, mentre avranno PID 5754, 5755, 5756… sull'host (Figura 3).

```
-- terminale 1: Avvio task da container
root@host:/# docker run -it ubuntu:18.04 bash
root@6a5aed60f2f9:/#
root@6a5aed60f2f9:/# watch ps ax &
[1] 10
root@6a5aed60f2f9:/# ps ax
    PID TTY      STAT   TIME   COMMAND
      1 pts/0    Ss     0:00   bash
     10 pts/0    T      0:00   watch ps ax
     11 pts/0    R+     0:00   ps ax

[1]+  Stopped    watch ps ax
root@6a5aed60f2f9:/#


-- terminale 2: Estrazione PID su host
root@host:/# ps ax | grep "watch ps ax" | grep -v grep
    5763 pts/0    T    0:00  watch ps ax
root@host:/#
```

Figura 3: PID di una task eseguita da container

1.3.5 Docker container default namespaces

Per comprendere la natura degli spazi di indirizzi all'interno di un container, risulta utile analizzare la traccia dell'esecuzione di containerd durante l'elaborazione del comando 'docker start C1', ottenuta per mezzo del comando '*strace*'.

```
root@host:/# strace -f -p $(pidof containerd) -o strace_log
root@host:/#
```

Figura 4: comando strace per tracciare il processo 'containerd'

Guardando la syscall '*unshare*', è possibile notare le seguenti flags [13]:

- CLONE_NEWNS: inizializza processo in una nuova mount namespace;
- CLONE_NEWUTS: inizializza processo in un nuovo UTS namespace;
- CLONE_NEWIPC: inizializza processo in un nuovo IPC namespace;
- CLONE_NEWNET: inizializza processo in un nuovo NET namespace;
- CLONE_NEWPID: inizializza processo in un nuovo PID namespace.

Figura 5: 'unshare' syscall di containerd

Si può osservare che non tutti i namespaces esistenti risultano separati dall'host.

Ciò è dimostrabile creando una nuova task interna al container, 'watch ps ax', per poi fare un paragone tra i namespaces della suddetta task e i namespaces della bash sull'host (Figura 6). Risultano:

- Namespaces isolati: ipc, mnt, net, pid, uts.
- Namespaces comuni all'host: cgroup, time, user.

La condivisione dello spazio d'indirizzi utente (user namespace) è osservabile anche nell'impostazione della syscall '*unshare*', dove manca la flag CLONE_NEWUSER. Quest'ultimo particolare porta delle considerazioni importanti sulla natura degli utenti all'interno dei container e sui privilegi in loro possesso.

```
root@host:/# ps ax | grep "watch ps ax" | grep -v "grep"
    10669 pts/1    S+     0:00 watch ps ax
root@host:/#
root@host:/# #ns di task interna al container
root@host:/# ls -lah /proc/10669/ns
totale 0
dr-x--x--x 2 root root 0 giu 26 15:08 .
dr-xr-xr-x 9 root root 0 giu 26 15:08 ..
lrwxrwxrwx 1 root root 0 giu 26 15:08 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 ipc -> 'ipc:[4026532258]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 mnt -> 'mnt:[4026532256]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 net -> 'net:[4026532261]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 pid -> 'pid:[4026532259]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 pid_for_children ->
'pid:[4026532259]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 time_for_children ->
'time:[4026531834]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 giu 26 15:08 uts -> 'uts:[4026532257]'
root@host:/#
root@host:/# #ns di task esterna al container (bash su host)
root@host:/#  ls -lah /proc/$$/ns
totale 0
dr-x--x--x 2 root root 0 giu 26 15:10 .
dr-xr-xr-x 9 root root 0 giu 26 13:58 ..
lrwxrwxrwx 1 root root 0 giu 26 15:10 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 net -> 'net:[4026531992]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 pid_for_children ->
'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 time_for_children ->
'time:[4026531834]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 giu 26 15:10 uts -> 'uts:[4026531838]'
root@host:/#
```

Figura 6: confronto tra namespaces di task da container e bash da host

1.4 Utenti e gruppi

1.4.1 Utente del container e utente dell'host

Gli utenti e i gruppi sono identificati rispettivamente dai codici UID e GID. In mancanza di un meccanismo di rimappatura di questi codici e data la condivisione del medesimo user namespace tra host e container, ne consegue un'importante considerazione: UID o GID uguali rispettivamente a UID o GID sull'host risultano equivalenti a questi, sia nel container che sull'host, in termini di privilegi.

Prendiamo per esempio l'utente '*root*' (UID=0, GID=0) : ha pieni privilegi di lettura, scrittura, esecuzione, ed è considerabile come un 'passpartout di sistema'.

In condizioni di default e in assenza di meccanismi di sicurezza, non esiste distinzione tra l'utente root del container e l'utente root dell'host: localmente e globalmente, sono di fatto lo stesso utente.

Ciò è interpretabile come una vulnerabilità: se, per assurdo, l'utente root di un container fosse in grado, nelle condizioni sopra descritte, di accedere alla cartella radice dell'host e di eseguire il comando '*unmount /*' , il sistema lo considererebbe un comando legittimo e, pertanto, lo porterebbe a termine, provocando un Denial-Of-Service.

1.4.2 Utente root in Docker container

Rimanendo nel contesto di Docker e nella sua configurazione base, l'utente predefinito nei containers è 'root' (UID=0, GID=0).

```
root@host:/# docker run -it --rm alpine:latest sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine

[ … ]
Status: Downloaded newer image for alpine:latest
/#
/# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(flo
ppy),20(dialout),26(tape),27(video)
```

Figura 7: utente root in Docker container

1.5 Storage drivers in Docker

1.5.1 Gestione mounts

La mount namespace permette di controllare l'accesso del container agli alberi file associati tramite il comando *mount*. Il Linux kernel regola la propagazione delle mount tramite una feature chiamata 'shared subtree' [14].

1.5.2 Volumi

I volumi sono un elemento Docker simile alle bind mounts ma con diversi vantaggi in termini sia di gestione che di performance. Alcune migliorie in confronto alle bind mounts sono [15]:

- Facile backup e migrazione;
- Gestione tramite Docker CLI e tramite Docker API;
- Interoperabilità dei volumi tra sistemi Linux e Windows;
- Facilità di condivisione tra containers;
- Contenuto cifrabile e altre funzionalità.

1.5.3 OverlayFS (Overlay Filesystem)

OverlayFS è un'*union mount filesystem*': si tratta di una feature del Linux kernel, presente dalla versione 4.0, che permette la creazione di un filesystem virtuale per mezzo della sovrapposizione di più cartelle [16].

L'albero di ciascuna cartella costituente è da intendere come un filesystem a sé stante. La procedura di 'union mount' prevede due cartelle sovrapponibili e una di lavoro:

- *lowerdir*: contiene i file read-only;
- *upperdir*: contiene i file accessibili in scrittura;
- *workdir*: cartella vuota necessaria al compimento delle operazioni di creazione del Overlay Filesystem [17].

La sovrapposizione delle cartelle *lowerdir* e *upperdir* porta alla creazione della cartella *merged* , che diventa l'effettivo punto di mount del container filesystem: essa provvederà a offrire una vista sommaria e interattiva dei due filesystem sottostanti.

- Se si provasse a creare un file o modificare un file read-only, il file risultante verrebbe creato nella cartella *merged* e nella cartella *upperdir* ;
- Se si provasse a cancellare un elemento read-only, l'elemento verrebbe eliminato dalla cartella *merged*, mentre verrebbe creato un file corrispondente a un 'segnaposto' nella cartella *upperdir*: si parla rispettivamente di '*whiteout file*' o di '*opaque directory*' a seconda se l'elemento sia un file o una directory [18].

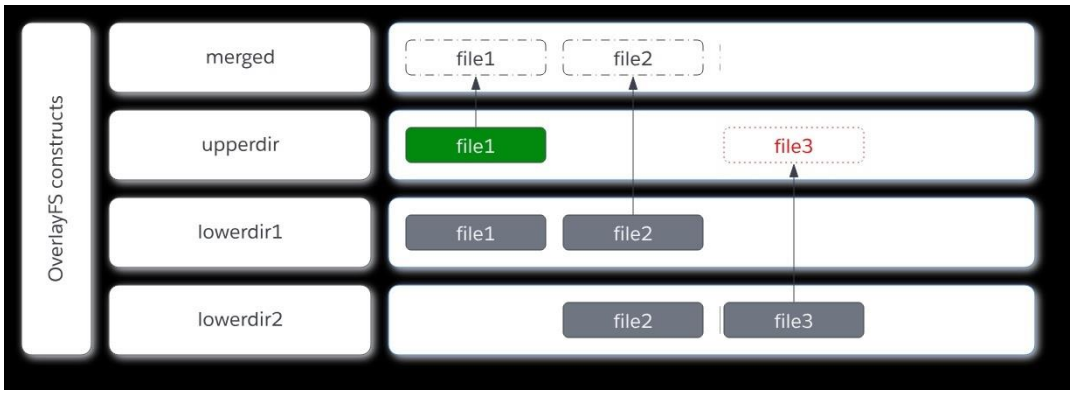

Figura 8: funzionamento di OverlayFS

1.5.4 OverlayFS in Docker container

Le immagini sono composte da una serie di layers che, tutti insieme, vanno a comporre la *lowerdir*: questa è rappresentata da una cartella (chiamata 'l') al cui interno si trova un riferimento simbolico, i 'symlinks', per ogni layer componente l'immagine. I layers dell'immagine seguono una struttura a nodi con relazione gerarchica, la cui radice è il layer 'di base' dell'immagine .

I layers dispongono delle seguenti cartelle e files, che possono far intuire una 'union mount' ricorsiva:

- *diff* : cartella contenente i files del layer;

- *lower*: escluso il livello più basso, contiene un riferimento al link del layer precedente, che funge da parente gerarchico;
- *work:* cartella di lavoro per OverlayFS;
- *merged*: cartella per union mount tra i contenuti del container e del parente;
- *link:* contiene riferimento symlink del layer attuale.

In cima all'immagine, per il meccanismo Copy-On-Write, si pone lo strato scrivibile del container. Questo strato contiene sempre gli elementi diff,lower,work,merged,link allo stesso modo dei layers superiori dell'immagine, con l'eccezione che questi costituiscono la 'union mount' finale per creare il filesystem [19]:

- *upperdir* è la cartella diff del livello container;
- *workdir* è la cartella work del livello container;
- *lowerdir* contiene, dal livello più alto al più basso, tutti i links agli strati dell'immagine;
- *merged* contiene la vista sommaria del container filesystem.

Dalla versione 23.0.0 del Docker Engine, il driver di archiviazione predefinito è *overlay2:* si è sostituito a *overlay* introducendo vantaggi quali il supporto nativo di *lowerdir* composte fino a 128 layers.

Dall'interno dei containers è possibile vedere gli hard links delle mount di overlayFS consultando, ad esempio, il file di configurazione '/etc/mtab'.

```
-- terminale1
root@host:/# docker run -it --rm ubuntu:18.04 bash
root@69bf355b1a2e:/#
root@69bf355b1a2e:/# cat /etc/mtab | grep overlay
overlay / overlay rw,relatime,lowerdir=/var/lib/docker/over-
lay2/l/AA7R6BL4AFZKXIQLGKKTX3ZVIW:/var/lib/docker/over-
lay2/l/GFEPLIZVGDMVASEVV2V3UV5O6Q,upperdir=/var/lib/docker/over-
lay2/059a7112faf893fffaabdc1a56b9ac73c35a0e73d745c60a97955c22150fdabd/
diff,workdir=/var/lib/docker/over-
lay2/059a7112faf893fffaabdc1a56b9ac73c35a0e73d745c60a97955c22150fdabd/
work 0 0
root@69bf355b1a2e:/#
root@69bf355b1a2e:/# touch ciao && ls
bin   ciao  etc   lib    media  opt   root  sbin  sys  usr
boot  dev   home  lib64  mnt    proc  run   srv   tmp  var
root@69bf355b1a2e:/#


-- terminale2
root@host:/# PATH_TO_OVERLAY=/var/lib/docker/overlay2
root@host:/#CONT_LAYER=059a7112faf893fffaabdc1a56b9ac73c35a0e73d745c60
a97955c22150fdabd
root@host:/#
root@host:/# # symlinks ai layers componenti il livello lowerdir
root@host:/# ls -al $PATH_TO_OVERLAY/l/ | grep
"GFEPLIZVGDMVASEVV2V3UV5O6Q\|AA7R6BL4AFZKXIQLGKKTX3ZVIW"
lrwxrwxrwx  1 root root    77 giu 30 13:23 AA7R6BL4AFZKXIQLGKKTX3ZVIW
->
../059a7112faf893fffaabdc1a56b9ac73c35a0e73d745c60a97955c22150fdabd-
init/diff
lrwxrwxrwx  1 root root    72 giu 12 18:25 GFEPLIZVGDMVASEVV2V3UV5O6Q
->
../6e9d0490d82626ac725c5397c89156b1091da8d317fccd95e1ff10d0ee3e7fbb/di
ff
root@host:/#
root@host:/# # livello merged
root@host:/# ls $PATH_TO_OVERLAY/$CONT_LAYER/merged
bin   ciao  etc   lib    media  opt   root  sbin  sys  usr
boot  dev   home  lib64  mnt    proc  run   srv   tmp  var
root@host:/#
root@host:/# # livello upperdir
root@host:/# ls $PATH_TO_OVERLAY/$CONT_LAYER/diff
ciao
root@host:/#
```

Figura 9: illustrazione di overlay2

1.6 Networking in Docker

Docker inserisce i container in una network dedicata. Tale network non è raggiungibile dall'esterno, a meno che non si configuri manualmente l'associazione tra le porte del container e le porte dell'host: essendo il net namespace del container separato dal net namespace dell'host, le interfacce network del container risultano indipendenti dalle interfacce network host, così come il suo indirizzo IP, le routing tables, il suo DNS solver e il resto dello stack network.

Si possono creare network personalizzate.

L'installazione di Docker Engine offre tre network predefinite:

- *none* network: i container all'interno di questa rete non hanno un IP assegnato, ma solo l'indirizzo di loopback. Pertanto, non hanno alcuna possibilità di operare in rete.
- *host* network: l'intero stack network dell'host sarà condiviso con i container che partecipano a questa rete;
- *bridge* network: di default, i container sono inseriti qui. L'indirizzo IP di rete è, di norma, 172.17.0.0, con submask 255.255.0.0. L'indirizzo 172.17.0.1 viene assegnato all'interfaccia host docker0, che assume il ruolo di bridge della network .

Ogni container all'interno della network ha un'interfaccia 'veth' collegata al bridge della propria network.

Per i container all'interno della network *bridge* è possibile comunicare con l'host in maniera bidirezionale, comunicare con altri containers e raggiungere la rete esterna alla macchina, come Internet. Rimangono comunque irraggiungibili dall'esterno, se non per mezzo di una porta associata con l'host.

Le network in docker hanno un DNS server incorporato che permette ai containers di risolvere i nomi degli atri containers per raggiungerli, anziché usare il loro IP.

Risulta inoltre possibile creare dei bridge tra le network per permettere a queste di comunicare tra loro, inserire firewalls e proxy servers [20].

1.7 Sicurezza predefinita in Docker

1.7.1 cgroups (control groups)

Questa funzionalità, offerta dal Linux kernel, permette la creazione di gruppi di controllo atti a limitare e gestire un certo gruppo di risorse quali CPU, uso della memoria, quantità di memoria, risorse network e operazioni I/O su dispositivi di blocco, per uno specifico gruppo di processi. Sono esposti nel container tramite uno pseudo-filesystem, che nei sistemi Linux più recenti corrisponde solitamente al percorso '/sys/fs/cgroup' [21].

Inoltre, un qualsiasi processo può accedere alla gerarchia dei propri cgroups: ad esempio, tramite il procfs pseudo-filesystem, al percorso '/proc/self/cgroup' [22].

Esistono due versioni dei cgroups, 'cgroup v1' e 'cgroup v2': le versioni più recenti di Docker utilizzano cgroup v2.

Le cgroups sono modificabili in fase di inizializzazione del container tramite opzioni su riga di comando: per il comando 'docker run' esistono opzioni come '--cpuset-cpus' o '--memory'.

A monte, i cgroups sono organizzati in slices: la slice in uso predefinito da Docker è 'system.slice', che è la stessa del sistema host [23]. Ne consegue che i gruppi di controllo utilizzati di default da Docker sono gli stessi presenti sull'host.

1.7.2 seccomp (Secure Computing)

Una feature del Linux kernel, usato di default in Docker dalla versione 1.10.0 [24], che permette di limitare le syscalls richiamabili dal namespace interno ai containers, impedendone l'esecuzione o filtrando gli argomenti ammessi per l'esecuzione.

A livello basso, questa feature opera sullo stato 'seccomp' del processo chiamante, basandosi su determinati profili di sicurezza che fungono da liste nere. in formato json.

Di default esiste già un profilo predefinito, 'default.json', per l'esecuzione standard; esiste inoltre l'opzione 'unconfined' per l'esecuzione senza restrizioni.

Per essere efficace, seccomp dev'essere prima di tutto abilitato dal kernel con CONFIG_SECCOMP=y [25].

1.7.3 AppArmor

Feature del Linux kernel che introduce un vincolo Mandatory Access Control per le operazioni da e verso un certo processo, attraverso l'associazione di una policy al container [26].

AppArmor ha funzionamento path-based: il profilo predefinito in Docker, ad esempio, vieta la scrittura su tutti i file figli diretti di '/proc', '/proc/sys' e l'utilizzo della syscall 'mount' [27].

Nonostante AppArmor sia divenuto standard nel modulo di sicurezza kernel, esso non è attivo di default in tutte le distribuzioni Linux: è standard, ad esempio, in Ubuntu e OpenSUSE, mentre va attivato nelle distribuzioni RHEL, Fedora e CentOS.

1.7.4 SELinux

Feature del Linux kernel che introduce un vincolo Mandatory Access Control per le operazioni da e verso un certo processo, attraverso l'associazione di policy al container.

La sicurezza implementata da SELinux segue il funzionamento label-based: estende le ACL di ogni file di sistema aggiungendo un tag con la forma '*user:role:type:level*' [28]:

- *User*: l'utente della policy che ha accesso a delle specifiche 'Roles', con un particolare 'Level'. Ogni utente Linux è mappato a un utente SELinux corrispondente, tramite una SELinux policy;
- *Role*: ereditabili, danno l'autorizzazione a certi 'Types';
- *Type*: riferiti a oggetti del filesystem o tipi di processo (quest'ultimi, nello specifico, sono anche detti 'domini'), servono alle SELinux policies per specificare come un certo 'Type' può accedere ad altri 'Types';
- *Level*: opzionale, livello di confidenzialità dell'informazione secondo il modello Bell-LaPaula, usato se SELinux è in modalità MLS(Multy-Layer Security) o MCS(Multy-Category Security). Queste modalità sono combinabili.

SELinux è attivo nei sistemi Fedora come alternativa ad AppArmor. Il demone di Docker può attivarne il supporto tramite la flag '—selinux-enabled'.

SELinux ha tre modalità:

- Enforcing: attiva le policy bloccanti;
- Permissive: le policy producono solo stampe di warnings;
- Disabled: SELinux policies disattivate.

1.7.5 Capabilities e processi privilegiati

I sistemi UNIX distinguono in linea generale due tipi di processo:

- I processi privilegiati (o 'processi root', UID=0), con pieni permessi;
- I processi non privilegiati (con UID≠0), i cui permessi dipendono dalle credenziali del processo quali GID e gruppi a cui appartiene.

Dalla versione 2.2 di Linux è stata introdotta una scomposizione dei privilegi che caratterizzano i processi root: le *capabilities*, ovvero 'capacità'. Ogni capability permette determinati comportamenti e operazioni all'interno di un processo, così come in un container Docker.

Un container Docker, infatti, è anche un processo che, come tale:

- Può esser eseguito come in modalità privilegiata, bypassando AppArmor e seccomp, con la flag '--privileged';
- Può esser eseguito con determinate capabilities abilitate, così da consentire solo specifiche operazioni privilegiate, facendo attenzione al profilo AppArmor e seccomp associati, che lavorano in maniera indipendente.

1.7.6 ACL delle componenti Docker

Generalmente, le componenti del framework Docker sono di proprietà dell'utente root (UID=0), mentre appartengono al gruppo 'docker' (il suo GID varia da sistema a sistema) o al gruppo root (GID=0).

```
root@host:/# ls -all /run/docker.sock
srw-rw---- 1 root docker 0 giu 21 15:48 /run/docker.sock
root@host:/#
```

Figura 10: ACL del Docker Unix socket (Docker API)

RIFERIMENTI

[1] Red Hat, “I vantaggi dei Container” [Online]. Available: https://www.redhat.com/it/topics/containers.

[2] Aqua, “Container Images: Architecture and Best Practices” [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-security/container-images/.

[3] “namespaces(7) - Linux manual page” [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[4] “Docker Overview | Docker Documentation” [Online]. Available: https://docs.docker.com/get-started/overview.

[5] “What Is Containerd? | Docker” [Online]. Available: https://www.docker.com/blog/what-is-containerd-runtime/.

[6] “containerd/containerd: An open and reliable container runtime” [Online]. Available: https://github.com/containerd/containerd.

[7] “Introducing runC: A lightweight universal container runtime | Docker” [Online]. Available: https://www.docker.com/blog/runc/.

[8] “dockercon-2016” [Online]. Available: https://github.com/crosbymichael/dockercon-2016/tree/master/.

[9] “containerd/runtime/v2/README.md at main” [Online]. Available: https://github.com/containerd/containerd/blob/main/runtime/v2/README.md.

[10] “Docker Engine API v1.43 Reference” [Online]. Available: https://docs.docker.com/engine/api/v1.43/.

[11] “About Storage Drivers | Docker Documentation” [Online]. Available: https://docs.docker.com/storage/storagedriver/.

[12] “docker create | Docker Documentation” [Online]. Available: https://docs.docker.com/engine/reference/commandline/create/.

[13] “clone(2) | Linux manual page” [Online]. Available: https://man7.org/linux/man-pages/man2/clone.2.html.

[14] “Building a container by hand using namespaces: The mount namespace” [Online]. Available: https://www.redhat.com/sysadmin/mount-namespaces.

[15] “Volumes | Docker Documentation” [Online]. Available: https://docs.docker.com/storage/volumes/.

[16] “Use The OverlayFS Storage Driver | Docker Documentation” [Online]. Available: https://docs.docker.com/storage/storagedriver/overlayfs-driver/.

[17] web.archive.org, “The Overlay Filesystem” [Online]. Available: https://web.archive.org/web/20220930060750/http://windsock.io/the-overlay-filesystem/.

[18] “kernel.org” [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[19] “docker/docs/userguide/storagedriver at main” [Online]. Available: https://github.com/tnozicka/docker/blob/master/docs/userguide/storagedriver/overlayfs-driver.md.

[20] “Networking Overview | Docker Documentation” [Online]. Available: https://docs.docker.com/network/.

[21] “Runtime metrics | Docker Documentation” [Online]. Available: https://docs.docker.com/config/containers/runmetrics/.

[22] “PROCFS /proc/self - IBM Documentation” [Online]. Available: https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=targets-procfs-procself.

[23] “dockerd | Docker Documentation” [Online]. Available: https://docs.docker.com/engine/reference/commandline/dockerd/#miscellaneous-options.

[24] “Docker Engine release notes | Docker Documentation” [Online]. Available: https://docs.docker.com/engine/release-notes/prior-releases/.

[25] “Seccomp security profiles for Docker” [Online]. Available: https://docs.docker.com/engine/security/seccomp/.

[26] “AppArmor --- The Linux Kernel Documentation” [Online]. Available: https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/apparmor.html.

[27] “moby/profiles/apparmor/template.go at master” [Online]. Available: https://github.com/moby/moby/blob/master/profiles/apparmor/template.go.

[28] “How SELinux separates containers using Multi-Level Security” [Online]. Available: https://www.redhat.com/en/blog/how-selinux-separates-containers-using-multi-level-security.