



**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

Relatore:
Chiar.mo Stefano Paraboschi

Matricola n.
1068142

Anno Accademico
2022/2023

ABSTRACT

I containers sono degli ambienti di lavoro e di runtime virtualizzati, divenuti popolari per la loro leggerezza, flessibilità, portabilità e la capacità di fornire sistemi isolati senza bisogno di hypervisor.

Essendo che si basano sul sistema operativo ospitante, tra host e container non vi è una completa separazione degli spazi d'indirizzi: senza le dovute misure di sicurezza, dunque, può aprirsi la possibilità di una escalation dei privilegi sulla macchina ospitante a partire dal container.

Docker è un framework popolare per la gestione dei container: semplifica ai developer l'interazione coi container e la "containerizzazione" delle applicazioni per mezzo di comandi di alto livello.

In questa tesi viene proposta un'illustrazione variegata di contesti di vulnerabilità atti a dimostrare, per mezzo di codici realizzati nei linguaggi Python3, C, Shell, Bash, Dockerfile, la possibilità di fuga da container, con lo scopo di aprire una riflessione su determinate vulnerabilità del framework Docker e le forme di mitigazione adoperabili.

INDICE DEI CONTENUTI

	pagina
1 INTRODUZIONE	0
1.1 Container.....	0
1.1.1 Container Management Framework	0
1.2 Isolamento processi in Linux	1
1.2.1 Spazi d'indirizzi.....	1
1.2.2 Gruppi di controllo	1
1.3 Sicurezza Linux	3
1.3.1 Moduli Capability.....	3
1.3.2 Secure Computing mode	3
1.3.3 Linux Security Modules	3
1.4 Docker.....	5
1.4.1 Architettura framework	5
1.4.2 Isolamento del container Docker	7
1.4.3 Immagine Docker	9
1.4.4 Filesystem del container Docker	9
1.4.5 Networking in Docker	10
1.4.6 Sicurezza del container Docker	11
1.5 Container Escape Vulnerability	12
RIFERIMENTI.....	13

INDICE DELLE FIGURE

	Pagina
Figura 1: comando docker run	6
Figura 2: unshare syscall di containerd.....	7
Figura 3: confronto ns processo container e bash host	8
Figura 4: OverlayFS in Docker.....	9

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.

I container forniscono, rispetto all'host, un minimo grado di isolamento dei processi attivi al loro interno per mezzo degli “spazi di indirizzi” (anche detti *namespace*): i processi interni al container creato “non hanno idea” di essere parte di un sottospazio dell'host.

L'isolamento offerto dai container non è totale: 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.

1.1.1 Container Management Framework

Esistono diversi framework per la gestione dei container, attraverso riga di comando o gestore applicativo. Generalmente, questi framework si distinguono in base al controllore o all'owner dei componenti del framework.

In base al controllo, si ha un framework *daemon-based* se il controllo delle richieste d'interazione con l'ambiente, i container e le immagini è gestito da un controllore centrale, detto “demone”; in caso contrario, il framework si dice *daemonless*.

Un esempio di framework *daemon-based* è il framework Docker, mentre un esempio di framework *daemonless* è il framework Podman [3].

In base all'owner dei componenti, si può distinguere il framework in *root* o *rootless* a seconda se l'owner sia rispettivamente l'utente con massimi privilegi del sistema operativo ospitante, detto `root`, oppure un utente non privilegiato.

1.2 Isolamento processi in Linux

Linux mette a disposizione diverse feature per l'esecuzione isolata di gruppi di processi, al fine di limitare l'accesso alle risorse di sistema o fornire un'ambiente di runtime virtualizzato.

1.2.1 Spazi d'indirizzi

Gli spazi d'indirizzi, anche detti *namespace*, sono una caratteristica dei kernel Linux: hanno il compito di isolare le risorse di un certo gruppo di processi dalle risorse di altri gruppi di processi [4].

Prendendo come esempio il caso specifico dei containers, i namespace sono responsabili dell'ambiente percepito dai processi interni al container: attraverso la virtualizzazione di un determinato gruppo di risorse host, ciascun namespace contribuisce alla creazione di un nuovo ambiente di lavoro, replicando un sistema operativo dentro al quale i processi del container vivono confinati. In tal modo, i processi interni al container non possono percepire altri processi esterni al container, né accedere a risorse non definite dai namespace.

Ad esempio, il net namespace gestisce le risorse network per un certo gruppo di processi: se un processo venisse originato con un net namespace diverso dall'analogo spazio d'indirizzi sull'host, come può essere dall'interno di un container, questo avrebbe l'impressione di avere uno stack network completamente indipendente dallo stack network "reale" dell'host [5].

Quando un certo namespace è condiviso con l'host, i processi fanno riferimento alle stesse risorse usate sull'host: se, per esempio, il gruppo di processi interni a un container non ha lo user namespace separato dall'host, allora l'utente `root` (UID 0) ed il gruppo `root` (GID 0) presenti nel container sono rispettivamente lo stesso utente e lo stesso gruppo definiti e utilizzati sull'host.

1.2.2 Gruppi di controllo

Questa funzionalità, offerta dal kernel Linux, 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 [6].

I cgroup sono organizzati in maniera gerarchica in elementi detti *unit* [7]:

- *Service unit*: consente il raggruppamento di più processi in un unico elemento di gestione;
- *Scope unit*: permette di raggruppare processi creati esternamente, come containers, sessioni utente, macchine virtuali;
- *Slice unit*: organizzano la gerarchia delle service unit e scope unit. Le slice sono a loro volta organizzate in una gerarchia.

Soffermendosi sulle slice, il sistema organizza in default quattro slice principali:

- La slice radice, *-.slice*;
- La slice per le macchine virtuali e container, *machine.slice*;
- La slice per le sessioni utente, *user.slice*;
- La slice per tutti i service di sistema, *system.slice*.

Un qualsiasi processo può accedere ai propri cgroup seguendo il percorso “/proc/self/cgroup” [8]. DalL’host, è possibile visionare la gerarchia dei gruppi di controllo al percorso “/sys/fs/cgroup”.

Oggi esistono due versioni di cgroup: v1 e v2 [9].

1.3 Sicurezza Linux

In generale, i sistemi UNIX e derivati, come Linux, distinguono due tipi di processo: i processi privilegiati (o “processi root”, UID=0), aventi pieni permessi di accesso, e i processi non privilegiati (con UID≠0).

Oltre a questa classificazione, il kernel Linux integra diversi sistemi di controllo dei permessi.

1.3.1 Moduli Capability

Dalla versione 2.2 di Linux, i processi non privilegiati hanno la possibilità di accedere a determinate risorse privilegiate per mezzo delle capability: singoli privilegi che caratterizzano i processi root [10]. Per esempio, la capability CAP_SYS_TIME permette ad un processo non privilegiato di impostare l’orologio di sistema.

1.3.2 Secure Computing mode

Il Secure Computing mode o Seccomp è uno strumento di sandboxing integrato nel kernel Linux dalla versione 2.6.12 [11].

Quando Seccomp è attivo su un processo, vengono consentiti ai suoi thread un limitato numero di syscall: read, write, exit, sigreturn.

Le versioni più recenti di Seccomp adottano il Berkeley Packet Filter per favorire una maggiore flessibilità nelle restrizioni: è possibile creare delle blacklist o whitelist per esplicitare, rispettivamente, le syscall proibite o le syscall consentite all’interno del processo [12].

A seconda di come viene impostato il filtro di Seccomp-bpf, richiamare una syscall non consentita da un processo sorvegliato può causare un segnale di risposta negativo, un logging dell’evento o la terminazione del processo [13].

1.3.3 Linux Security Modules

Il Linux Security Modules è un framework che integra diversi sistemi di sicurezza basati sul paradigma Mandatory Access Control, cioè sistemi che applicano restrizioni d’accesso alle risorse per mezzo di policy: due esempi sono SELinux e AppArmor [14].

SELinux segue il funzionamento label-based: estende le ACL di ogni file di sistema aggiungendo un tag con la forma ‘*user:role:type:Level*’ [15].

- *User*: l'utente della policy che ha accesso a delle specifiche "Role", con un particolare "Level". Ogni utente Linux è mappato a un utente SELinux corrispondente, tramite una policy SELinux;
- *Role*: ereditabili, danno l'autorizzazione a certi "Type";
- *Type*: riferiti a oggetti del filesystem o tipi di processo (quest'ultimi, nello specifico, sono anche detti domini), servono alle policy SELinux per specificare come un certo "Type" può accedere ad altri "Type";
- *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 ha tre modalità di gestione delle policy: Enforcing, Permissive, Disabled, la cui risposta a un accesso interdetto corrisponde rispettivamente a blocco, stampa d'avvertimento, indifferenza.

Mentre SELinux è solitamente attiva nei sistemi Fedora-based, AppArmor rappresenta l'alternativa operante più comune nei sistemi OpenSUSE e Debian-based.

Le policy di AppArmor sono delle whitelist che definiscono i privilegi con cui il processo può accedere alle risorse di sistema e ad altri processi, garantendo così una forma di protezione verso il processo, oltre al controllo del suo comportamento [16].

AppArmor organizza le policy in profili: ciascun profilo basa il proprio controllo su un determinato dominio, come un applicativo, un utente o l'intero sistema.

Ogni profilo può agire in tre modalità: *enforced*, *complain* o *unconfined*, dove la trasgressione del profilo causa, rispettivamente, il blocco, il logging, o l'esecuzione incondizionata dell'operazione.

I profili usano le *rule* per definire l'accesso a determinate capability, risorse network, files: tali *rule* sono applicate a insiemi di percorsi file per mezzo di espressioni regolari [17].

1.4 Docker

Docker è un framework open-source, sviluppato dalla Docker Inc., per lo sviluppo e l'impiego di container basati su kernel Linux [18]. Offre una gestione di alto livello dei containers, sia da riga di comando che da applicativo desktop.

1.4.1 Architettura framework

Docker è daemon-based e, di base, root: è sviluppato come un'architettura client-server conforme alle regole REST API: la comunicazione tra clienti e controllore, basata su HTTP, è gestita in maniera stateless seguendo un formato standard specificato nella documentazione della Docker API [19].

Di base, l'architettura Docker è così costituita:

- lato client, la *Docker CLI*: dotato interfaccia a riga di comando 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* [20]: anche detto Docker Daemon, resta in ascolto di default su un socket UNIX, 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 socket TCP;
- *containerd* [21] [22]: 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;
- *runC* [23]: 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, capability. Ha completo supporto dei Linux namespace, inclusi user namespace: è responsabile della creazione dei namespace ed esecuzione dei

containers. In particolare, runC è invocata da containerd-shim [24]: 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 [25].

Si prenda per esempio l'esecuzione del comando da CLI `docker run` (Figura 1):

1. tramite richiesta API, viene ordinato a dockerd la creazione del container con l'immagine selezionata che, qualora mancante, verrà scaricata dal registry;
2. ricevuta tramite API la richiesta di inizializzazione del container, dockerd riferisce a containerd di preparare l'ambiente d'esecuzione ed avviare il container.

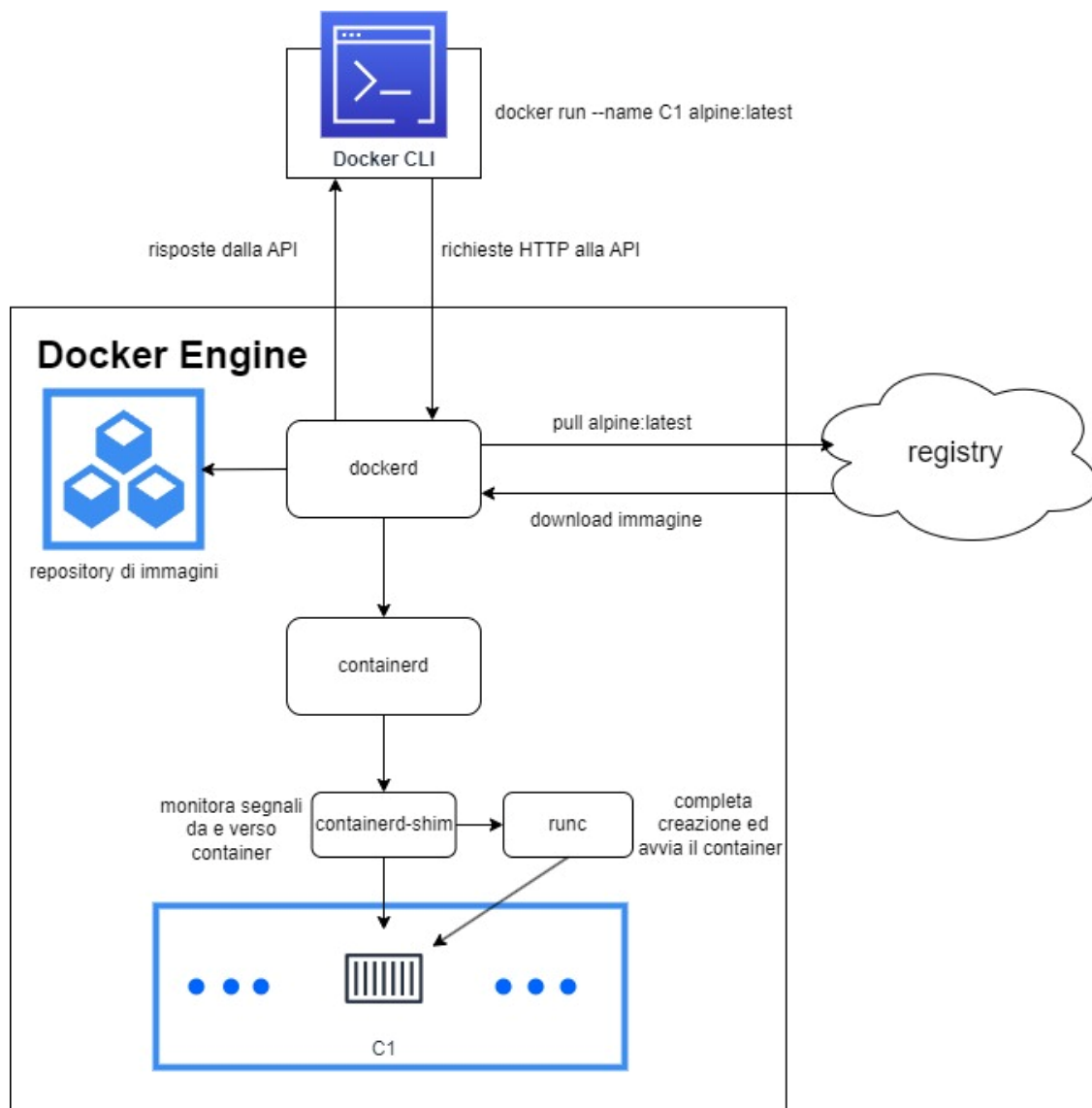


Figura 1: comando `docker run`

1.4.2 Isolamento del container Docker

Gli spazi d'indirizzi vengono creati in fase di inizializzazione del container dal componente runC. Più nel dettaglio, runC esegue la syscall *unshare* per la creazione dei namespace interni al container [26], poi effettua una fork del processo di init (PID 1) all'interno del container e, infine, termina la propria esecuzione.

Essendo runC componente di containerd, è possibile ottenere la traccia delle syscalls necessarie alla creazione dei namespace applicando il comando *strace* su containerd. In particolar modo, si può notare l'impostazione delle flag di *unshare* (Figura 2) per la creazione di nuovi spazi d'indirizzi del container come, ad esempio, *CLONE_NEWPID* [27], responsabile della creazione del nuovo pid namespace, ovvero lo spazio d'indirizzi che permette una numerazione PID indipendente dalla numerazione PID “reale” sull'host.

```
6146 prctl(PR_SET_NAME, "runc:[1:CHILD]") = 0
6141 <... close resumed>                  = 0
6146 unshare(CLONE_NEWNS|CLONE_NEWUTS|CLONE_NEWIPC|CLONE_NEWNET|CLONE_NEWPID <unfinished ...>
6141 read(9, <unfinished ...>
6146 <... unshare resumed>                = 0
```

Figura 2: *unshare* syscall di containerd

Un modo più diretto per verificare la separazione degli spazi d'indirizzi è realizzabile confrontando la bash di un terminale sull'host con un processo in loop generato da un container (Figura 3).

Il contenuto della sottocartella di processo *ns* presenta dei symlinks che fanno riferimento ai namespace del processo, identificabili univocamente grazie all'inode mostrato nelle parentesi quadre [28].

Si possono estrarre le informazioni dei namespace del processo terminale applicando il comando `ls -l /proc/$$/ns`. Inoltre, se si crea una task “lunga” in un container e si ricerca la task su host tramite il comando *ps*, è possibile estrarre il PID “reale” della task containerizzata e accedere alle informazioni della sua sottocartella *ns* come fatto per il terminale.

In questo modo, è possibile distinguere quali namespace di un container Docker risultano indipendenti dal sistema operativo ospitante e quali non: in condizioni di default, risultano isolati i namespace *ipc*, *mnt*, *net*, *pid*, *uts*, mentre il namespace *cgroup* risulta condiviso o meno se è attivo, rispettivamente, *cgroup v1* o *cgroup v2* [29].

```

root@host:/# docker run -it ubuntu:18.04 bash
root@d819d5713825:/# watch ps ax & 2>/dev/null ps ax
[1] 10
    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@d819d5713825:/#

----aprendo un altro terminale sull'host
root@host:/# ps ax | grep "watch ps ax" | grep -v grep
    5149 pts/0        T            0:00 watch ps ax
root@host:/# ls -l /proc/5149/ns
totale 0
lrwxrwxrwx 1 root root 0 lug 19 15:35 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 ipc -> 'ipc:[4026532257]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 mnt -> 'mnt:[4026532255]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 net -> 'net:[4026532260]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 pid -> 'pid:[4026532258]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 pid_for_children ->
'pid:[4026532258]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 time_for_children ->
'time:[4026531834]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 lug 19 15:35 uts -> 'uts:[4026532256]'
root@host:/# ls -l /proc/$$/ns
totale 0
lrwxrwxrwx 1 root root 0 lug 19 15:37 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 net -> 'net:[4026531992]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 pid_for_children ->
'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 time_for_children ->
'time:[4026531834]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 lug 19 15:37 uts -> 'uts:[4026531838]'

```

Figura 3: confronto ns processo container e bash host

Il cgroup attivo determina, inoltre, il cgroup driver di Docker, che in cgroup v1 è il cgroupfs /docker, mentre in cgroup v2 è la slice di sistema system.slice.

Di default, i cgroup driver sono creati sotto il cgroup root (il cgroup driver “/”).

Tramite comando, è possibile configurare i namespace del container e i cgroup in fase d’inizializzazione del container: ad esempio, aggiungendo al comando docker run opzioni come --pid o --memory [30].

1.4.3 Immagine Docker

Ogni container Docker è creato sulla base di un oggetto Docker detto *immagine*: una configurazione read-only contenente le dipendenze necessarie alla creazione dell'ambiente di runtime del container.

Le immagini sono strutturate in layers che, creati in successione, aggiungono file, cartelle, librerie, informazioni di configurazione all'ambiente.

Docker dà la possibilità agli sviluppatori di creare nuove immagini personalizzate a partire da un'immagine pre-esistente grazie al Dockerfile: un file di configurazione per la progettazione di immagini, facente uso di istruzioni con sintassi specifica come FROM per "importare" un'immagine base, RUN per eseguire comandi, USER per selezionare l'utente attivo nell'esecuzione delle istruzioni e, successivamente, nel container [31].

Per completare la realizzazione dell'immagine, la Docker CLI mette a disposizione il comando `docker build` [32].

1.4.4 Filesystem del container Docker

Il filesystem montato nel container Docker è, di default, un *union mount filesystem* basato su una feature chiamata OverlayFS [33], presente nel kernel Linux dalla versione 4.0.

Il filesystem di un container Docker è formato da tre livelli (Figura 4):

- *lowerdir*: livello read-only del filesystem;
- *upperdir*: livello scrivibile del filesystem;
- *merged*: risultato dalla procedura di *union mount* dei precedenti livelli, viene montato nel container. Esso contiene i riferimenti per ogni elemento presente in *upperdir* ed ogni elemento presente in *lowerdir* ma assente in *upperdir*.



Figura 4: OverlayFS in Docker

La creazione del filesystem del container segue un meccanismo Copy-On-Write: viene creato un layer scrivibile dedicato al container, dove il livello *upperdir* è rappresentato dalla cartella `diff` e il livello *merged* da una cartella omonima. La

lowerdir è costituita dai layer dell'immagine adottata, dei quali il container memorizza i riferimenti simbolici [34].

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.

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*, mentre la cancellazione di un elemento read-only ne determina l'eliminazione del riferimento dalla cartella *merged*, mentre verrebbe creato un file corrispondente a un "segnaposto" nella cartella *upperdir* [35].

Mentre il filesystem del container è generalmente volatile, Docker mette a disposizione dei meccanismi di memorizzazione persistente montabili nel container, detti volumi: sono un elemento Docker simile alle bind mounts ma con diversi vantaggi, come la facilità di backup, migrazione, condivisione e interoperabilità [36].

Un volume può esser montato durante l'esecuzione del comando `docker run` inserendo l'opzione `-v` a cui seguono, in successione e separati da carattere ":", due elementi obbligatori e uno opzionale: rispettivamente, il percorso su host indicante il volume da montare, il percorso nel filesystem del container indicante il punto di mount e l'opzione `ro` qualora si volesse rendere non scrivibile il volume montato.

1.4.5 Networking in Docker

Docker inserisce i container in una rete dedicata.

Le network stesse sono degli oggetti Docker: la Docker CLI mette a disposizione il comando `docker network` per la creazione, rimozione, gestione di queste [37].

L'installazione di Docker Engine offre tre network predefinite:

- Network *none* [38] : 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.
- Network *host* [39] : l'intero stack network dell'host sarà condiviso con i container che partecipano a questa rete;
- Network *bridge* [40] : 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 server DNS incorporato che permette ai containers di risolvere i nomi degli altri 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 server proxy [41] .

La Docker CLI permette di selezionare la rete in cui inizializzare il container durante l'esecuzione del comando `docker run` tramite l'opzione `--net` [42].

1.4.6 Sicurezza del container Docker

Docker possiede dei profili di default per Seccomp e AppArmor, applicati ai container in esecuzione quando non viene specificata un'altra policy: ad esempio, la Docker CLI offre l'opzione `--security-opt` per applicare una policy personalizzata.

Normalmente, i container Docker vengono eseguiti come processi non privilegiati, con un set limitato di capability, ma è possibile rimuovere o aggiungere tutte le capability messe a disposizione dal kernel. Per esempio, la Docker CLI permette l'inizializzazione di containers privilegiati, con tutte le capability, usando l'opzione `--privileged`, mentre è possibile gestire le capability con le opzioni `--cap-add` e `--cap-drop`.

Esiste una policy SELinux ad-hoc per Docker, il cui supporto può essere attivato dal demone `dockerd` tramite la flag `--selinux-enabled`.

1.5 Container Escape Vulnerability

Questa vulnerabilità descrive generalmente la capacità di un utente malevolo di poter effettuare azioni privilegiate sull'host a partire da un container.

Per Docker, possiamo generalmente individuare tre scenari d'attacco:

- Attacco esterno, allo scopo di penetrare i servizi esposti in rete;
- Accesso a un container compromesso, dov'è possibile sfruttare le proprietà d'ambiente per raggiungere l'host;
- Insider malevolo: un utente di sistema non privilegiato che tenta di accedere a privilegi non previsti dal suo profilo.

Un container Docker presenta vulnerabilità ad attacchi esterni quando è possibile, dalla rete esterna, individuare le vulnerabilità per mezzo di testing sui servizi esposti: a seguito di questa prima fase, detta *enumerazione* [43], si può aprire la possibilità di un accesso imprevisto al container, detto *initial foothold* [44].

Lo studio dell'ambiente containerizzato può far emergere determinate caratteristiche che l'attaccante può sfruttare per ottenere l'accesso a risorse di privilegio superiore, cioè una *elevazione dei privilegi* (o *vertical privilege escalation*, [45]).

Relativamente alla macchina ospitante, un'elevazione dei privilegi massima risulta nell'accesso completo ad ogni risorsa root di sistema.

L'elevazione dei privilegi può costituire obiettivo anche per gli utenti non privilegiati di sistema, purchè Docker sia accessibile senza necessità di privilegi.

Non tutti gli attacchi hanno le stesse caratteristiche: mentre l'abuso di risorse può esser una strategia di lungo termine e difficile da individuare, l'interruzione del servizio Docker è rilevabile in fretta, con alto impatto a breve termine.

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] “A Qualitative and Quantitative Analysis of Container Engines”, [Online]. Available: <https://arxiv.org/pdf/2303.04080.pdf>.
- [4] “namespaces(7) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [5] “Linux namespaces - Wikipedia”, [Online]. Available: https://en.wikipedia.org/wiki/Linux_namespaces.
- [6] “Chapter 1. Introduction to Control Groups”, [Online]. Available: https://access.redhat.com/documentation/it-it/red_hat_enterprise_linux/7/html/resource_management_guide/chap-introduction_to_control_groups.
- [7] “1.2 Default Cgroup Hierarchies Red Hat Enterprise Linux 7 | Red Hat”, [Online]. Available: https://access.redhat.com/documentation/it-it/red_hat_enterprise_linux/7/html/resource_management_guide/sec-default_cgroup_hierarchies.
- [8] “PROCFS /proc/self - IBM Documentation”, [Online]. Available: <https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=targets-procfs-procself>.
- [9] “Understanding the new control groups API [LWN.net]”, [Online]. Available: <https://lwn.net/Articles/679786/>.
- [10] “capability(7) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man7/capability.7.html>.

- [11] “Security/Sandbox/Seccomp - MozillaWiki”, [Online]. Available:
<https://wiki.mozilla.org/Security/Sandbox/Seccomp>.
- [12] “seccomp(2) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man2/seccomp.2.html>.
- [13] “A seccomp overview [LWN.net]”, [Online]. Available:
<https://lwn.net/Articles/656307/>.
- [14] “Linux Security Module Usage --- The Linux Kernel Documentation”, [Online].
Available: <https://www.kernel.org/doc/html/latest/admin-guide/LSM/index.html>.
- [15] “How SELinux separates containers using Multi-Level Security”, [Online]. Available:
<https://www.redhat.com/en/blog/how-selinux-separates-containers-using-multi-Level-security>.
- [16] “QuickProfileLanguage Wiki AppArmor / AppArmor”, [Online]. Available:
<https://gitlab.com/apparmor/apparmor/-/wikis/About>.
- [17] “QuickProfileLanguage Wiki AppArmor / AppArmor”, [Online]. Available:
<https://gitlab.com/apparmor/apparmor/-/wikis/QuickProfileLanguage#file-rules>.
- [18] “Docker Overview | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/get-started/overview>.
- [19] “Develop with Docker Engine API |² Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/api/>.
- [20] “dockerd | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/reference/commandline/dockerd/>.
- [21] “What Is Containerd? | Docker”, [Online]. Available:
<https://www.docker.com/blog/what-is-containerd-runtime/>.
- [22] “containerd/containerd: An open and reliable container runtime”, [Online]. Available:
<https://github.com/containerd/containerd>.
- [23] “Introducing runC: A lightweight universal container runtime | Docker”, [Online].
Available: <https://www.docker.com/blog/runc/>.
- [24] “dockercon-2016”, [Online]. Available: <https://github.com/crosbymichael/dockercon-2016/tree/master/>.

- [25] “containerd/runtime³e/v2/README.md at main”, [Online]. Available: <https://github.com/containerd/containerd/blob/main/runtime/v2/>.
- [26] “unshare(2) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man2/unshare.2.html>.
- [27] “clone(2) | Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man2/clone.2.html>.
- [28] “namespaces(7) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [29] “Runtime metrics | Docker Documentation”, [Online]. Available: <https://docs.docker.com/config/containers/runmetrics/>.
- [30] “Docker run reference | Docker Documentation”, [Online]. Available: <https://docs.docker.com/engine/reference/run/>.
- [31] “Best practices for writing Dockerfiles | Docker Documentation”, [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.
- [32] “Dockerfile reference | Docker Documentation”, [Online]. Available: <https://docs.docker.com/engine/reference/builder/>.
- [33] “Use The OverlayFS Storage Driver | Docker Documentation”, [Online]. Available: <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>.
- [34] “docker/docs/userguide/storagedriver at main”, [Online]. Available: <https://github.com/tnozicka/docker/blob/master/docs/userguide/storagedriver/overlayfs-driver.md>.
- [35] “kernel.org”, [Online]. Available: <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>.
- [36] “Volumes | Docker Documentation”, [Online]. Available: <https://docs.docker.com/storage/volumes/>.
- [37] “docker network | Docker Documentation”, [Online]. Available: <https://docs.docker.com/engine/reference/commandline/network/>.
- [38] “None network driver | Docker Documentation”, [Online]. Available: <https://docs.docker.com/network/drivers/none/>.

- [39] “Host network driver | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/network/drivers/host/>.
- [40] “Bridge network driver | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/network/drivers/bridge/>.
- [41] “Networking Overview | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/network/>.
- [42] “Network Containers | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/tutorials/networkingcontainers/>.
- [43] “Browser Information Discovery, Technique T1217 - Enterprise | MITRE ATT&CK”,
[Online]. Available: <https://attack.mitre.org/techniques/T1217/>.
- [44] “Initial Access, Tactic TA0001 - Enterprise | MITRE ATT&CK”, [Online]. Available:
<https://attack.mitre.org/tactics/TA0001/>.
- [45] “Privilege Escalation, Tactic TA0004 - Enterprise | MITRE ATT&CK”, [Online].
Available: <https://attack.mitre.org/tactics/TA0004/>.