
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

# INDICE DELLE FIGURE

Pagina

**Non è stata trovata alcuna voce dell'indice delle figure.**

# 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.

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 gestione dei namespace

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 [3].

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.

Ogni namespace può esser visto come una virtualizzazione di un certo aspetto dell'ambiente, identificabile e condivisibile tra processi appartenenti allo stesso ambiente di lavoro [4].

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.

Con la stessa logica, un nuovo spazio d'indirizzi di mount (mnt namespace) può dare l'impressione a un certo gruppo di processi di avere un proprio filesystem indipendente, con cartella radice propria, mentre un pid namespace diverso dall'analogo namespace dell'host può illudere un processo di avere PID 1, quando il suo "reale" PID nel namespace host è ben più alto, dato che il PID 1 è già assegnato al processo di inizializzazione del sistema ospitante.

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.

## 1.3 Docker

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 [5].

Docker è daemon-based e, di base, root. Esso 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 state-less seguendo un formato standard specificato nella documentazione della Docker API [6].

Docker è disponibile anche come applicativo desktop.

### 1.3.1 Architettura framework

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* [7] : 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* [8] [9]: 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* [10]: 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 namespace, inclusi user namespace: è responsabile della creazione dei namespace ed esecuzione dei

containers. In particolare, runC è invocata da containerd-shim [11]: 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 [12].

# 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] «Linux namespaces - Wikipedia,» [Online]. Available: https://en.wikipedia.org/wiki/Linux_namespaces.

[5] «Docker Overview | Docker Documentation,» [Online]. Available: https://docs.docker.com/get-started/overview.

[6] «Develop with Docker Engine API | Docker Documentation,» [Online]. Available: https://docs.docker.com/engine/api/.

[7] «dockerd | Docker Documentation,» [Online]. Available: https://docs.docker.com/engine/reference/commandline/dockerd/#miscellaneous-options.

[8] «What Is Containerd? | Docker,» [Online]. Available: https://www.docker.com/blog/what-is-containerd-runtime/.

[9] «containerd/containerd: An open and reliable container runtime,» [Online]. Available: https://github.com/containerd/containerd.

[10] «Introducing runC: A lightweight universal container runtime | Docker,» [Online]. Available: https://www.docker.com/blog/runc/.

[11] «dockercon-2016,» [Online]. Available: https://github.com/crosbymichael/dockercon-2016/tree/master/.

[12] «containerd/runtime/v2/README.md at main,» [Online]. Available: https://github.com/containerd/containerd/blob/main/runtime/v2/.