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 Limitazione risorse in kernel Linux

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 [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 [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.

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

1.2.3 Capacità e processi privilegiati

1.3 Linux Security Modules

1.3.1 Seccomp

1.3.2 AppArmor

1.3.3 SELinux

1.4 Docker

Docker è un framework open-source, sviluppato dalla Docker Inc., per lo sviluppo e l'impiego di container containers basati su kernel Linux [5]. 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. 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].

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].

Per capire le dinamiche dell'architettura, si prenda per esempio il comando `docker run` impartito da docker CLI. L'operazione consta di due fasi:

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

1.4.2 Namespace 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 [13], 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* per la creazione di nuovi spazi d'indirizzi del container come, ad esempio, CLONE_PID [14], responsabile della creazione del nuovo pid namespace, ovvero lo spazio d'indirizzi che permette una numerazione PID indipendente dalla numerazione PID "reale" sull'host.

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:

1. dopo aver creato il processo X all'interno del container, si estrae il "vero" PID del processo X sull'host e si estraggono i namespace relativi visualizzando le

informazioni della cartella “/proc/vero_pid_X/ns” in formato esteso, dove “vero_pid_X” andrebbe sostituito col PID precedentemente estratto sull’host;

2. con la bash in uso sull’host, allo stesso modo del punto precedente, si estraggono i namespace visualizzando le informazioni della cartella “/proc/$$/ns” in formato esteso, dove $$ è il PID del processo chiamante attualmente in uso, cioè la bash.

Ognuno dei punti sopra elencati mostra il contenuto della cartella `ns` appartenente al processo preso in considerazione: dentro, sono presenti dei symlinks che fan riferimento ai namespace del processo, che sono identificabili grazie all’inode mostrato nelle parentesi quadre [15].

In questo modo, è possibile distinguere quali namespace di un container risultano indipendenti dal sistema operativo ospitante e quali non. In particolare, per un container Docker in configurazione di default, risultano:

- Namespace isolati: ipc, mnt, net, pid, uts;
- Namespace condivisi con l’host: cgroup, time, user.

Tramite comando, è possibile specificare in fase d’inizializzazione la natura dei namespace dei container: ad esempio, è possibile condividere il pid namespace dell’host con il container aggiungendo l’opzione `--pid=host` al comando `docker run` impartito da CLI [16].

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, sovrapposti, aggiungono informazioni di configurazione quali files, cartelle, variabili d’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 che, per mezzo di istruzioni con sintassi specifica, permette la progettazione, creazione e personalizzazione di immagini.

Per completare la realizzazione dell’immagine, la Docker CLI mette a disposizione il comando `docker build` [17] .

1.4.4 OverlayFS in Docker

Il filesystem montato nel container Docker è un *union mount filesystem*, basato su una feature chiamata OverlayFS [18] , presente nel kernel Linux dalla versione 4.0: tale feature permette la creazione di un filesystem non-persistente per mezzo della sovrapposizione di diversi livelli del filesystem:

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

Perché la procedura di *union mount* sia completata, viene disposta inoltre una cartella di lavoro chiamata *workdir*.

La creazione del filesystem del container segue un meccanismo Copy-On-Write: viene creato uno strato 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 da ogni layer dell'immagine adottata, dei quali il container memorizza i riferimenti simbolici [19].

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*.

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.

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/.

[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/.

[13] «unshare(2) - Linux manual page,» [Online]. Available: https://man7.org/linux/man-pages/man2/unshare.2.html.

[14] «clone(2) | Linux manual page,» [Online]. Available: https://man7.org/linux/man-pages/man2/clone.2.html.

[15] «namespaces(7) - Linux manual page,» [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[16] «Docker run reference | DOcker Documentation,» [Online]. Available: https://docs.docker.com/engine/reference/run/.

[17] «Dockerfile reference | Docker Documentation,» [Online]. Available: https://docs.docker.com/engine/reference/builder/.

[18] «Use The OverlayFS Storage Driver | Docker Documentation,» [Online]. Available: https://docs.docker.com/storage/storagedriver/overlayfs-driver/.

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