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

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.

Soffermandosi 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 containers 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: 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).

Docker 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`:

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.

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* per la creazione di nuovi spazi d'indirizzi del container come, ad esempio, CLONE_PID [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.

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

Prima di tutto, si estraggono le informazioni dei namespace del processo terminale applicando il comando `ls -l /proc/$$/ns` . Dopodichè, si crea una task "lunga" in un container e si ricerca la task su host tramite il comando `ps`, per poi estrarre il PID "reale" della task 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].

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:

- *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*. Questa costituisce il punto di mount del filesystem presente nel container.

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 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 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 [43] e AppArmor [44], 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 prorietà 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 vulnerabiltà per mezzo di testing sui servizi esposti: a seguito di questa prima fase, detta *enumerazione* [45], si può aprire la possibilità di un accesso imprevisto al container, detto *initial foothold* [46].

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,* [47]).

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.

2 IMPLEMENTAZIONE

2.1 Accesso non privilegiato a docker

Far parte del gruppo ‘docker’ permette l’utilizzo di Docker senza necessariamente far parte del file ‘Sudoers’, il file dei super users.

Si ipotizzi che un insider malevolo voglia acquisire il ruolo di utente `root` sull’host; perché ciò sia possibile, l’attaccante deve:

- Far parte del gruppo “docker”;
- Avere accesso alla binary “docker” della Docker CLI;
- Avere accesso a Docker root-based.

L’attacco richiede due soli passi. Nel primo passo, si crea un container da Docker CLI, col comando `docker run` seguito dai parametri:

- Opzione “-it”: viene aperto lo standard input sul container e allocata sessione terminale;
- Opzione “-v /:/host”: monta la cartella radice “/” dell’host, sottoforma di volume, al percorso “/host” del filesystem interno al container;
- Opzione “--privileged” : Seccomp e AppArmor disabilitati, tutti i moduli capability vengono abilitati per il processo container;
- Opzione “--net=host”: il container viene creato con accesso al network stack dell’host;
- Opzione “--pid=host”: il pid namespace è lo stesso dell’host. Questo significa che, dal container, è possibile vedere tutti i processi presenti nel namespace dell’host, ad esempio tramite comando `ps a`.

Il secondo passo richiede l’attivazione della syscall *chroot*, utilizzabile grazie alla capability CAP_SYS_CHROOT: questa syscall cambia il riferimento base per la risoluzione dei percorsi file all’interno del container. Specificando come argomento di chiamata il volume montato, verrà preso come riferimento base la cartella radice di sistema.

Il risultato finale è una shell interattiva sul filesystem dell’host, con propagazione permanente delle modifiche e privilegi `root`.

```
dev@host:~$ docker images
REPOSITORY              TAG       IMAGE ID       CREATED        SIZE
alpine                  latest    5e2b554c1c45   8 weeks ago    7.33MB

dev@host:~$ docker run -it -v /:/host --rm --privileged --pid=host
--net=host alpine sh

/ # chroot /host

root@host:/# touch /rootfile
root@host:/# exit

/ # exit

dev@host:~$ cd /
dev@host:/$ ls -all
totale 80

[...]

-rw-r--r--   1 root root     0  8 lug 11.05 rootfile

[...]

dev@host:/$ rm rootfile
rm: rimuovere il file regolare vuoto protetto dalla scrittura
'rootfile'? s
rm: impossibile rimuovere 'rootfile': Permesso negato

dev@host:/$
```

Figura 1: P.O.C. usando binary docker

2.2 Misconfiguration della capability CAP_SYS_PTRACE

Assegnare troppi privilegi root ad un certo container Docker può portare alla configurazione di un container compromesso.

La seguente dimostrazione vuole illustrare uno scenario dove un container ha abilitato CAP_SYS_PTRACE, capability che permette il tracciamento e debugging dei processi in esecuzione entro i namespace del container, per mezzo della syscall *ptrace*.

Versioni più recenti del kernel Linux hanno integrato una forma di protezione dall'abuso di *ptrace*, predefinendo dei limiti al suo campo d'azione: si può considerare l'esempio di Ubuntu che, a partire dalla versione 10.10, consente l'uso di *ptrace* solo verso i processi figli [48].

Si prenda quindi come riferimento un sistema operativo host Ubuntu superiore alla versione 10.10: dall'interno di un container, per tracciare i processi figli sarebbe necessario consentire la syscall ptrace, la cui esecuzione è bloccata da Seccomp e AppArmor.

Per semplicità, si decida quindi di non selezionare alcun profilo Seccomp nè AppArmor, impostando entrambi i profili in modalità *unconfined*, così da togliere le restrizioni su ptrace.

Per effettuare un debugging, ptrace ha bisogno di agganciarsi ad un processo di cui è specificato il PID: se il pid namespace fosse condiviso con l'host, risulterebbe possibile dall'interno di un container la visualizzazione di tutti i processi del namespace host, ma sarebbe possibile l'aggancio dei soli processi aventi stesso UID del processo tracciante.

Per ottenere il ruolo di root sulla macchina, quindi, occorre che l'utente attivo all'interno del container sia `root`, con UID 0.

Intercettando un processo attivo sull'host, come potrebbe essere un processo server, è possibile iniettare del codice malevolo con istruzioni compilate in linguaggio macchina, così da dirottare l'esecuzione regolare delle istruzioni ed imporre la creazione di un processo che resti in attesa di una connessione remota su una determinata porta host.

Noto l'indirizzo dell'interfaccia host, si può accedere al sistema operativo ospitante in qualità di utente `root`.

2.2.1 Exploit ptrace_infect.py

2.3 Abuso del User Mode Helper

Il kernel Linux mette a disposizione un'interfaccia, detta User Mode Linux, che funge da kernel per lo spazio utente, introducendo così uno strato di separazione dall'effettivo kernel Linux destinato alle interazioni di basso livello e sensibili [49].

A supporto delle feature kernel del User Mode Linux, viene messo a disposizione lo User Mode Helper (abbreviato, UMH), un programma che permette al kernel l'esecuzione di chiamate di sistema nello spazio utente. Queste chiamate di sistema sono fornite da processi che fan uso delle funzioni del UMH (quali *call_usermodehelper, call_usermodehelper_exec*) [50].

2.3.1 Abuso del cgroup-v1 release_agent

La feature cgroup-v1 ha un'opzione che fa uso del UMH: quando abilitata, la terminazione di tutti i processi attivi nel cgroup richiede al UMH di eseguire la routine presente in release_agent [51] [52].

L'esecuzione di questo attacco richiede:

- Possedere privilegi `root` per poter accedere ai root cgroup;
- Possedere un container con sufficienti privilegi per montare un cgroup, facendo uso della syscall *mount*: occorre quindi abilitare la capability CAP_SYS_ADMIN, mentre vanno disabilitati Seccomp e AppArmor.

All'interno del container, viene montato il root cgroup di sistema, contenente il release_agent. Dentro alla cartella del cgroup radice, viene creata una cartella che automaticamente monterà un nuovo cgroup figlio.

Dentro al cgroup figlio vi è notify_on_release: scrivendo 1 dentro a tale file, verrà abilitato il supporto del release_agent.

Si prepara un file eseguibile contenente un programma malevolo: per la dimostrazione, si è scelto come payload una reverse shell che connette a un host remoto in attesa per una sessione bash.

Tale file eseguibile è destinato al release_agent, che verrà eseguito dal kernel; tuttavia, il kernel prende come riferimento base per la risoluzione dei percorsi la cartella radice dell'host.

Se il container usa overlayFS, allora il file malevolo creato si troverà nel livello upperdir: è possibile individuare il path assoluto verso la cartella di sistema montata come upperdir grazie al file di configurazione /etc/mtab [53].

Estratto il percorso upperdir per raggiungere il file eseguibile, si sovrascrive il contenuto del release_agent con la stringa risultante.

Per azionare il meccanismo, occorre inserire un processo veloce nel cgroup.procs del figlio: una volta finito il processo, il kernel attiverà il payload e connetterà l'host remoto alla macchina ospitante.

2.3.2 Abuso del core_pattern

Il root fileystem del container monta lo pseudo-filesystem /proc: un'interfaccia verso le strutture dati del kernel. Mentre la maggior parte del /proc filesystem è montato come read-only, alcuni dei file sono modificabili: nella sottodirectory /proc/sys/kernel, sono presenti dei file che permettono l'impostazione di diversi parametri del kernel, tra cui core_pattern [54].

Il core_pattern viene utilizzato dal kernel nella procedura di creazione del core dump, un file contenente lo stato della memoria al momento della terminazione imprevista di un certo programma [55].

Nel core_pattern, è possibile specificare la destinazione del core dump o, qualora la stringa contenuta nel core_pattern iniziasse con il carattere "|", di specificare un programma da eseguire in user space.

Alla terminazione imprevista di un processo, viene richiesto al UMH di leggere il core_pattern per completare la procedura di creazione del core dump [56].

All'interno del container, normalmente, il filesystem /proc è completamente read-only, ma esistono configurazioni in cui è consentita la scrittura del core_pattern da parte di `root`: ad esempio, il caso in cui il filesystem /proc è montabile, con la capability CAP_SYS_ADMIN abilitata e le strutture del LSM framework disabilitate, come il caso in cui il container è completamente privilegiato.

Quando il core_pattern è scrivibile da `root`, si procede con la preparazione di un file malevolo eseguibile, si estrae il suo percorso nel livello upperdir, estraendo i dati necessari da /etc/mtab e lo si inserisce nel core_pattern, precedendolo col carattere "|".

Si crea un file la cui esecuzione deve terminare in maniera anomala: una volta eseguito, il messaggio "segmentation fault (core dumped)" indicherà sia completamento della procedura di core dump che, dunque, dell'esecuzione del file malevolo.

2.4 Abuso dei symlink di processo

Nel pseudo-filesystem /proc, le risorse dei rispettivi processi sono raggiungibili presso le sottocartelle intitolate coi relativi PID.

Tra queste risorse, ci sono dei riferimenti simbolici a file o cartelle, come il symlink root, che specifica la radice del processo, o il symlink exe, contenente il percorso file al comando eseguito per inizializzare il processo.

Altri symlink sono presenti direttamente nella radice di /proc, come ad esempio /proc/self: un riferimento simbolico alla cartella col PID del processo che vi sta accedendo. Un processo che accede ai contenuti di /proc/self, dunque, sta accedendo ai contenuti della sottocartella che ha il suo stesso PID.

Quando un processo tenta l'accesso ad un riferimento simbolico nella struttura /proc, il kernel parte dalla cartella radice associata al processo per risolvere il percorso indicato.

2.4.1 Abuso del symlink root

Prendendo come esempio il cgroup-v1 release_agent, è possibile procedere con un approccio brute-force per indovinare, sull'host, il PID di un processo interno al container e accedere al suo symlink root, che si risolverà con la cartella radice del sistema containerizzato.

Sapendo che /proc ha la medesima struttura in ogni filesystem presente sul sistema, si può inserire nel release_agent il percorso "/proc/[PID]/root/payload" dove [PID] è una variabile che partirà da 1 (o più, dato che il PID 1 certamente non può essere un processo interno al container) per arrivare al massimo PID di sistema, mentre "payload" è il file malevolo creato nel container.

2.4.2 Abuso del processo "runC init"

In Docker, i comandi `docker run` e `docker exec` consentono l'esecuzione di un processo all'interno di un container: tali processi sono responsabilità di runC, che prima crea il processo figlio *runC init*, assegna a questo le dovute restrizioni (come i namespace), e lo avvia nel container. Infine, *runC init* chiama la syscall *execve* per eseguire la binary richiesta: ad esempio, /bin/bash o /bin/sh per una sessione terminale nel container [57].

Traendo vantaggio dalle funzionalità del symlink */proc/self*, è possibile spingere il processo `runC init` ad eseguire il proprio symlink *exe*, indicante la binary */usr/sbin/runC* presente sul sistema, così da inizializzare un'istanza di sé stesso.

Ciò è possibile manipolando una binary che potrebbe esser richiamata da un processo in entrata.

L'esecuzione di /proc/self/exe porta due conseguenze significative:

- La registrazione di /proc/self/exe tra i processi attivi, rendendo estraibile il PID del chiamante `runC init`, ad esempio tramite snapshot fornito da comando `ps`;
- L'apertura del descrittore file di runC, il cui riferimento verrà quindi registrato nella sottocartella *fd* del processo `runC init`, nella struttura /proc.

L'attaccante dovrà quindi estrarre il PID del `runC init`, accedere alle sue risorse nel pseudo-filesystem /proc e aprire il riferimento al descrittore file di runC in scrittura, così da sovrascrivere la binary runC con un nuovo contenuto eseguibile quale, ad esempio, una reverse shell: nelle versioni di Docker Engine inferiori alla 18.09.2, ciò è possibile [58].

La binary runC è di proprietà di `root`, quindi è necessario possedere UID 0 nel container per poter effetttuare l'operazione.

Il processo `runC init` non si risolverà in una binary consentita, quindi terminerà in maniera imprevista. Nell'intervallo di tempo che va dall'esecuzione del symlink *exe* da parte di `runC init` fino alla sua eliminazione, l'attaccante ha l'opportunità di sfruttare una race condition cosicchè, alla prossima esecuzione di un comando Docker facente uso della componente runC, sarà possibile l'esecuzione privilegiata di codice arbitrario sula macchina ospitante.

2.5 Abuso del Docker socket

Il socket `docker.sock` è responsabile della gestione della comunicazione con dockerd tramite il servizio RESTful di Docker API. I comandi impartiti, ad esempio, da Docker CLI arrivano a questo componente per mezzo di messaggi con protocollo HTTP.

2.5.1 Docker UNIX socket montato

L'attacco descritto a sezione 2.1 può esser realizzato anche nello scenario di un container compromesso, a patto che questo sia stato creato montando come volume `docker.sock`: tale mount è responsabile della comunicazione con dockerd tramite la Docker API. Occorre, inoltre, che l'utente del container sia root o abbia accesso alla binary docker.

Uno scenario di container compromesso con le condizioni di cui sopra può esistere con la configurazione Docker-in-Docker, usata in fase di Continuous Integration per progetti che richiedono il testing di immagini Docker [59].

Esistono, tuttavia, altri scenari dove è necessaria montare la UNIX socket: un esempio sono alcuni applicativi per la gestione centralizzata dell'ambiente Docker, come Traefik [60].

Per precauzione, è uso montare `docker.sock` in read-only così da impedire la propagazione delle modifiche sul componente presente in host.

2.5.2 Docker TCP socket esposto

Docker offre l'opzione di comunicare con dockerd dall'esterno, esponendo il Docker socket tramite l'associazione con una porta TCP.

Di default, assegnare una porta TCP a docker.sock rende la comunicazione con la porta associata di tipo HTTP, non protetta.

L'esposizione del servizio di Docker API aumenta la superficie d'attacco con lo scenario di attacco esterno.

2.5.3 Exploit: dockerio.py

Questo script semplifica la comunicazione con la Docker API tramite riga di comando a scopo di testing. In particolare, è possibile utilizzare tale script in un container compromesso per completare una fuga dall'ambiente Docker per completare un host takeover.

Lo script permette di effettuare comandi simili alla Docker CLI:

- Comando run: analogo a `docker run`;
- Comando exec: esegue un determinato processo in un container. In particolare, è possibile approfondire la richiesta con:
    - Opzione `--revsh`: il containter apre una reverse shell verso una socket di destinazione;
    - Opzione `--it-cmd`: rende possibile aprire una sessione con terminale interattivo nel container, simile a `-it` ma senza attach;
- Comando ps: mostra tutti i container e informazioni relative. In particolare, è possibile approfondire la richiesta con:
    - Opzione `rm` *[id_container]*: ferma e rimuove il container;
    - Opzione *[id_container]*: mostra le informazioni del solo container specificato;
- Comando image: comandi per la gestione di una singola immagine. In particolare, la richiesta è da approfondire con:
    - Opzione `load` *[nome_cont] [sorgente_http]*: permette di caricare da indirizzo remoto sorgente_http un'immagine derivata da un container in formato compresso '.tar' e di assegnargli un nome nome_cont;
    - Opzione `rm` *[id_immagine]*: rimuove una immagine;
    - Opzione *[id_immagine]*: mostra informazioni singola immagine
- Comando images: mostra tutte le immagini;
- Comando info: fornisce informazioni relative al Docker Engine e all'host;
- Comando event: fornisce una cronologia degli eventi.

Ognuno di questi comandi è l'astrazione di una o più richieste alla restFUL API.

Ogni richiesta è composta quindi da un percorso HTTP obbligatorio, seguito eventualmente da una searchQuery o da un JSON che specificasse i parametri della richiesta. Le richieste possono quindi essere GET per l'estrazione dati, DELETE per l'eliminazione degli elementi e POST per tutte le altre.

Questo script segue gli standard descritti nella Docker API v1.43 Documentation.

Lo script consente di interagire con uno UNIX socket o con un TCP socket, per mezzo di un oggetto 'session', gestore di alto livello per una sessione HTTP.

Per interagire con un UNIX socket, questo va montato nella session tramite un adattatore HTTP: l'oggetto UnixAdapter svolge questo compito. L'oggetto UnixAdapter incapsula ricorsivamente altre due classi, UnixConnectionPool e UnixConnection, al fine di specificare '/run/docker.sock' come UNIX socket a cui 'parlare' per ottenere una comunicazione con Docker API e, di conseguenza, con dockerd.

- UnixConnection eredita da urllib3.HTTPConnection, la quale consente la gestione di una singola sessione HTTP;
- UnixConnectionPool eredita da urllib3.HTTPConnectionPool, la quale consente la gestione di una pool di connessioni HTTP persistenti;
- UnixAdapter eredita da requests.adapters.HTTPAdapter ed è responsabile per l'implementazione di un'interfaccia per il livello Transport [61].

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 Analisys 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] «capabilities(7) - Linux manual page,» [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.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 | 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/runtime/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] «Seccomp security profiles for Docker,» [Online]. Available: https://docs.docker.com/engine/security/seccomp/.

[44] «AppArmor security profiles for Docker | Docker Documentation,» [Online]. Available: https://docs.docker.com/engine/security/apparmor/.

[45] «Browser Information Discovery, Technique T1217 - Enterprice | MITRE ATT&CK,» [Online]. Available: https://attack.mitre.org/techniques/T1217/.

[46] «Initial Access, Tactic TA0001 - Enterprise | MITRE ATT&CK,» [Online]. Available: https://attack.mitre.org/tactics/TA0001/.

[47] «Privilege Escalation, Tactic TA0004 - Enterprise | MITRE ATT&CK,» [Online]. Available: https://attack.mitre.org/tactics/TA0004/.

[48] «SecurityTeam/Roadmap/KernelHardening - Ubuntu Wiki,» [Online]. Available: https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection.

[49] «User Mode Linux HOWTO --- THe Linux Kernel Documentation,» [Online]. Available: https://www.kernel.org/doc/html/v5.9/virt/uml/user_mode_linux.html.

[50] «linux/kernel/umh.c at master - torvalds/linux,» [Online]. Available: https://github.com/torvalds/linux/blob/master/kernel/umh.c.

[51] «Control Groups --- The Linux Kernel documentation,» [Online]. Available: https://www.kernel.org/doc/html/v5.14/admin-guide/cgroup-v1/cgroups.html.

[52] «linux/kernel/cgroup/cgroup-v1.c at master - torvalds/linux,» [Online]. Available: https://github.com/torvalds/linux/blob/master/kernel/cgroup/cgroup-v1.c.

[53] gnu.org, «mtab,» [Online]. Available: https://www.gnu.org/software/hurd/hurd/translator/mtab.html.

[54] «proc(5) --- Linux manual pages,» [Online]. Available: https://web.archive.org/web/20160303182044/http://manpages.courier-mta.org/htmlman5/proc.5.html.

[55] «core(5) - Linux manual page,» [Online]. Available: https://man7.org/linux/man-pages/man5/core.5.html.

[56] «linux/fs/coredump.c at master - torvalds/linux,» [Online]. Available: https://github.com/torvalds/linux/blob/master/fs/coredump.c.

[57] «Breaking out of Docker via RunC -- Explaining CVE-2019-5736,» [Online]. Available: https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/.

[58] «DOcker Engine 18.09 release notes | Docker Documentation,» [Online]. Available: https://docs.docker.com/engine/release-notes/18.09/#18092.

[59] «Continuous Integration with Docker | Docker Documentation,» [Online]. Available: https://docs.docker.com/build/ci/.

[60] «traefik - Official Image | Docker Hub,» [Online]. Available: https://hub.docker.com/_/traefik.

[61] «requests.adapters --- Requests 2.31.0 documentation,» [Online]. Available: https://requests.readthedocs.io/en/latest/_modules/requests/adapters/.