POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Sicurezza dei container Linux

Relatori
Prof. Antonio Lioy
Ing. Andrea Atzeni
Ing. Paolo Smiraglia

Candidato
Agostino Palomba

Luglio 2018

Sommario

In questo elaborato verranno analizzate le problematiche presenti nel contesto della sicurezza informatica e relative alla tecnologia dei container. Nei primi capitoli, verrà quindi effettuata un'analisi dello stato dell'arte, dove si compareranno i vari tipi di virtualizzazione basati sulle macchine virtuali e sui container, mettendo inoltre a confronto i punti di forza e gli svantaggi se presenti, per ognuna di queste tecnologie di virtualizzazione. In seguito, sulla base del materiale fruibile online, verranno presentate le principali tecnologie di gestione dei container ad oggi maggiormente utilizzate, descrivendo dunque per ciascuna di esse l'architettura, il funzionamento e le varie implementazioni di sicurezza native e non. In seguito si andranno ad elencare e descrivere alcuni dei più recenti attacchi portati alle varie tecnologie di gestione dei container, con riferimento ad una tabella più esaustiva presente nel sito del *Common Vulnerabilities and Exposures*(CVE) la quale presenterà tutte le vulnerabilità scoperte fino ad oggi, per poi arrivare ad una visione comparativa della sicurezza presente in ambito container.

A seguito dell'introduzione e della descrizione delle varie tecnologie e delle vulnerabilità ad esse associate, dopo aver presentato i vari punti di forza che ciascuna di queste tecnologie fornisce nell'ambito della sicurezza e a seguito di una descrizione dei vari meccanismi di sicurezza che ad oggi sono presenti per tale scopo, ci si focalizzerà sulla presentazione di una nuova soluzione di sicurezza basata sull'automatizzazione di policy SELinux, prendendo dunque come esempio implementativo un ambiente basato su container multipli che dovranno comunicare tra di loro, attraverso l'implementazione di una rete dedicata ad essi, ed in modo sicuro.

Più precisamente si andrà a gestire tramite assegnazione di policy di sicurezza, la configurazione di regole per il controllo/filtraggio del traffico di rete all'interno dell'intero sistema con lo scopo di veicolare il traffico di rete e quindi l'inoltro stesso dei pacchetti. Questo in sostanza servirà a definire quale di questi container potrà comunicare con chi ed a quale di questi dovrà essere negato l'accesso diretto verso un altro specifico container.

Verrà inoltre gestita la parte di isolamento dei container rispetto al sistema operativo in esecuzione sulla macchina host, limitando quindi l'accesso a file e directory specifici all'interno del sistema operativo in esecuzione sulla macchina host, oltre a limitare le risorse hardware associate a ciascun container .

In conclusione verrà fatta una valutazione d'impatto sia in termini di sicurezza sia in termini di prestazioni.

Indice

Capitolo 1

Concetto di container e varie implementazioni

In questo capitolo si andrà ad introdurre e descrivere il concetto di container, partendo con un' analisi delle sostanziali differenze presenti rispetto alla virtualizzazione classica, per poi arrivare a descrivere le componenti, i casi d'uso e le varie tecnologie di gestione dei container che ad oggi risultano essere maggiormente utilizzate.

1.1 Introduzione ai Container e differenze con la virtualizzazione classica

La virtualizzazione si riferisce ad un meccanismo di astrazione delle risorse hardware con lo scopo di metterle a disposizione per l'esecuzione di uno specifico software. Tale tecnologia viene generalmente utilizzata all'interno di data center ed in particolar modo all'interno delle infrastrutture IT.

D'altronde la comodità di una IT "elastica" è data dal fatto che a seconda del variare del carico di lavoro un'impresa potrà attivare e disattivare nuove macchine virtuali, quindi in pratica attivare o disattivare nuovi server con lo scopo di poter permettere una gestione più dinamica.

Questo approccio alla virtualizzazione è tecnicamente chiamato anche con il nome di full virtualization, che è nato ormai da diversi anni e si è diffuso velocemente in vari ambiti, tanto che oggi è anche possibile installare piattaforme di virtualizzazione su singoli PC e far convivere su di essi sistemi operativi differenti.

La peculiarità della full virtualization sta nel fatto che la singola macchina virtuale è completamente separata e autonoma rispetto alle altre. Quando essa si attiva, si viene a definire la controparte virtuale di un server fisico, quindi anche tutte le sue risorse quali: processori, storage, connessioni di rete e sistema operativo, che vengono "mappate" in maniera trasparente sulle risorse fisiche che l'hardware sottostante può mettere a disposizione.

L'applicazione eseguita da una macchina virtuale non riesce a vedere nessuna differenza tra l'ambiente virtuale in cui si trova e un vero e proprio server fisico. A garantire la trasparenza di questa emulazione vi è l'hypervisor. Tale strumento rappresenta uno strato software che si occupa di attivare e disattivare le macchine virtuali e di tradurre le chiamate delle applicazioni alle funzioni e alle risorse del loro sistema operativo in altre chiamate alle risorse fisiche effettivamente disponibili. A seconda dei prodotti e degli approcci adottati per la virtualizzazione, un hypervisor viene eseguito direttamente dall'hardware del server fisico o da un sistema operativo convenzionale.

Un esempio si può basare sull'adottare un server virtuale in un ambiente per lo sviluppo di applicativi software o per il test delle applicazioni, in cui il punto non è tanto quello di creare

un'infrastruttura dinamica, quanto quello di andare a creare degli ambienti completamente separati fra di loro. La full virtualization infatti, consente ad esempio la coesistenza di più macchine virtuali basate su sistemi operativi del tutto differenti.

In altri casi però, le macchine virtuali in genere, sono considerate come un approccio eccessivamente pesante, ed una buona alternativa ad essi sono i container, una forma di virtualizzazione a livello di sistema operativo che si è diffusa più di recente e che è stata in gran parte adottata per i servizi di cloud computing. Un container si può ancora considerare come un server virtualizzato ma solo per lo spazio utente, ossia la parte virtualizzata è l'ambiente di esecuzione delle applicazioni e non tutti i componenti sottostanti (dal sistema operativo in giù verso l'hardware). Il sistema operativo e il kernel dello stesso sono quindi in comune a tutti i container avviati sulla macchina host. In questo approccio non esiste alcun hypervisor, ma è presente solo un sistema che "impacchetta" le applicazioni o più in generale i servizi applicativi in container, gestendone l'attivazione e la disattivazione dei container stessi e creando un livello di astrazione fra questi e il sistema operativo che li ospita. Uno dei punti di forza dei container è che sono molto più leggeri delle macchine virtuali in genere in termini di utilizzo di risorse hardware, perché non devono necessariamente avere un loro sistema operativo dedicato. Così facendo, sono molto più veloci da avviare e sospendere, quindi più indicati per quegli ambienti in cui il carico di elaborazione varia sensibilmente e in maniera non prevedibile a priori. Nella figura 1.1 viene illustrato un grafico di confronto tra la virtualizzazione classica e quella basata su container.

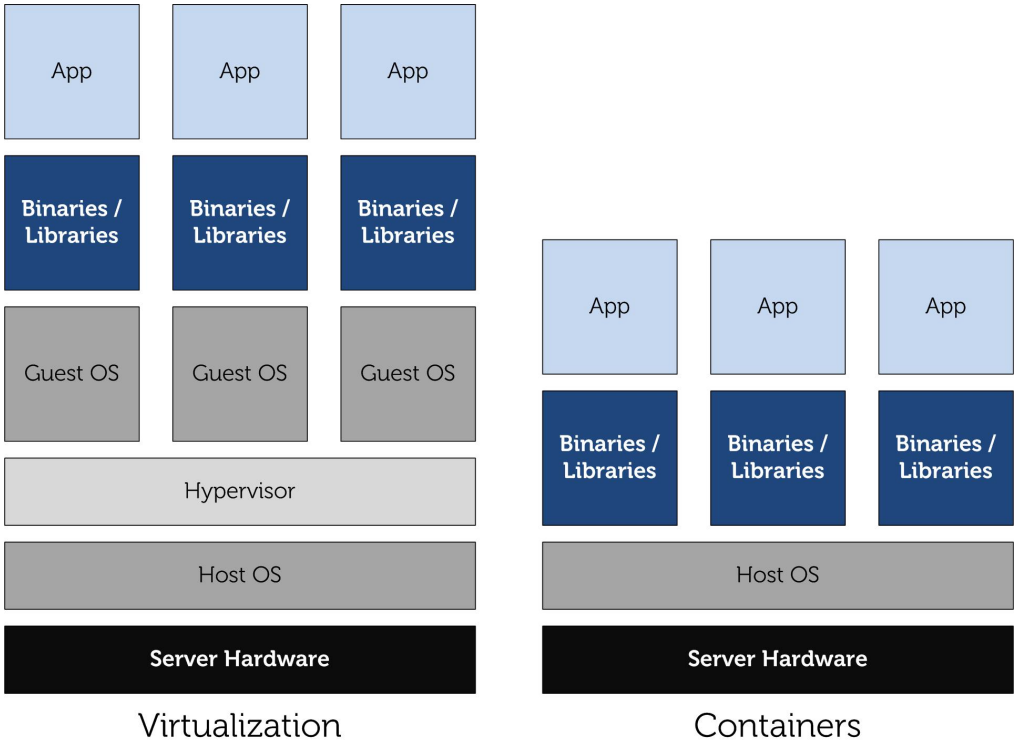

Figura 1.1. Container vs Virtual Machine [10] .

1.2 Hypervisor (Virtual Machine Manager)

L'Hypervisor o virtual machine manager è un software che fa uso delle tecniche di virtualizzazione hardware al fine di emulare un ambiente di calcolo per differenti sistemi operativi, che tra loro condivideranno una singola macchina host. L'Hyervisor risulta quindi essere un sottile strato che si interpone tra l'hardware della macchina sottostante (o il sistema operativo se eventualmente presente sulla macchina host) e gli uno o più i sistemi operativi installati sulla le macchina virtuale. Ad ogni sistema operativo installato sulla macchina virtuale vengono dedicate a priori risorse

hardware quali processore, memoria spazio di archiviazione, oltre ad una o più interfacce di rete. In questo modo ogni sistema operativo avrà le proprie risorse hardware allocate staticamente ed il proprio kernel, ovvero il kernel del sistema operativo installato su macchina virtuale. Questo metodo permette un alto livello di isolamento tra la macchina host e le virtual guest machine.

Il virtual machine monitor (es. Xen, open source monitor di macchine virtuali, VMware ESX attuale standard di virtualizzazione in ambito enterprise o Hyper-V monitor di macchine virtuali di Microsoft) deve operare in maniera trasparente senza pesare con la propria attività sul funzionamento e sulle prestazioni dei sistemi operativi.

Svolge quindi attività di controllo al di sopra di ogni sistema, permettendone lo sfruttamento anche come monitor e debugger delle attività dei sistemi operativi e delle applicazioni in modo da scoprire eventuali malfunzionamenti, al fine di poter intervenire celermente. I requisiti richiesti per questo scopo sono quelli di compatibilità, performance e semplicità. Gli ambiti di applicazione delle macchine virtuali sono molteplici ed eterogenei fra loro, poiché la virtualizzazione ad oggi rappresenta una componente utile alla realizzazione della sicurezza.

L'hypervisor può dunque controllare ed interrompere eventuali attività pericolose. Tra l'altro, i vantaggi della virtualizzazione che i sostenitori della tecnologia vedono in questa soluzione, quando questa è ben progettata, sono dovuti alla riduzione e al controllo dei costi grazie al consolidamento dell'hardware. L'hypervisor infatti, può allocare le risorse dinamicamente quando e dove necessario, ridurre in modo drastico il tempo necessario alla messa in opera di nuovi sistemi, isolare l'architettura nel suo complesso da problemi a livello di sistema operativo ed applicativo, abilitare ad una gestione più semplice di risorse eterogenee e, come già accennato, facilitare il collaudo ed il debugging di ambienti controllati. Esistono due tipi di Hypervisor, che sono il Bare Metal Hypervisor e l'O.S Based Hypervisor, come mostrato in figura 1.2.

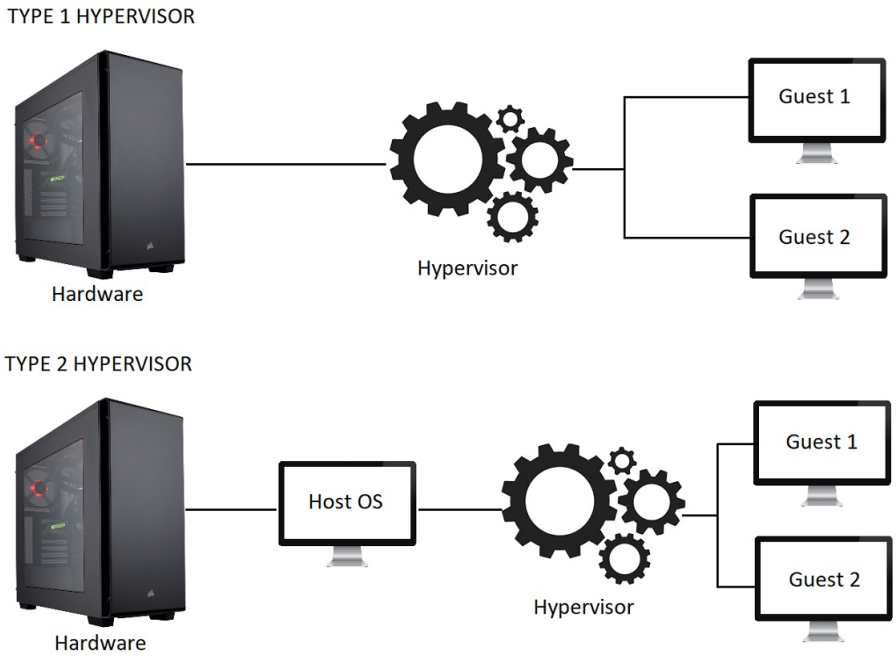

Figura 1.2. Hypervisor.

Bare Metal Hypervisor

Questo tipo di Hypervisor come descritto nella figura 1.3, viene avviato direttamente sull'host hardware per la gestione dei guest OS ed il controllo dell'hardware stesso. Un esempio di questa tipologia sono: VMware ESXi, vSphere Hypervisor, Microsoft Hyper-V, Citrix XenServer, Oracle VM, KVM, IBM z/VM.

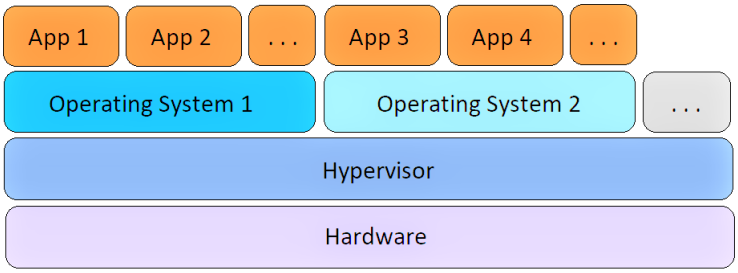

Figura 1.3. Bare Metal Hypervisor.

O.S Based Hypervisor

L'Hypervisor di tipo O.S based, descritto nella figura 1.4, viene avviato come un qualunque programma dal sistema operativo. Esso ha il compito di astrarre il sistema operativo della macchina host dai sistemi operativi guest. Un esempio di questa tipologia sono: VMware Workstation, VMware Player, Oracle VirtualBox.

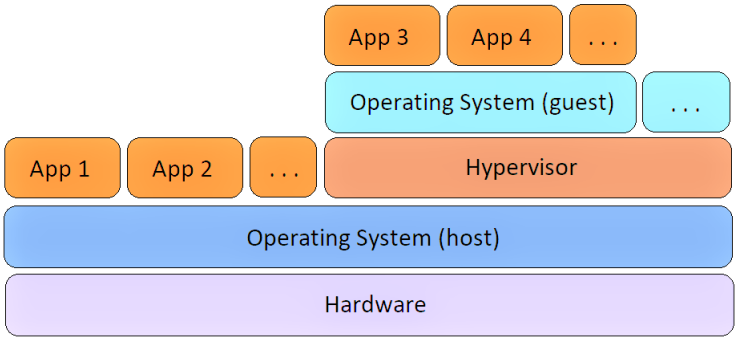

Figura 1.4. O.S. Based Hypervisor.

1.3 Container

Come accennato in precedenza, i container rappresentano degli spazi utente isolati in esecuzione sul sistema operativo di un server o macchina host in genere. Si tratta quindi di funzionalità del sistema operativo Linux, che permettono di virtualizzare alcune risorse di sistema per eseguire diverse applicazioni in ambienti operativi eterogenei, all'interno di uno stesso server fisico, invece di ricreare virtualmente tutta la macchina, ovvero risorse processore, risorse storage, risorse di rete e sistema operativo, come avviene invece quando si utilizza un approccio basato sulle virtual machine. Viene quindi astratto il solo ambiente di esecuzione delle applicazioni con i suoi settaggi fondamentali. I container dunque, eseguono le istruzioni dell'applicazione direttamente sulla CPU, senza richiedere alcun meccanismo di emulazione (come, invece, avviene per le macchine virtuali), permettendo di risparmiare risorse e attuando una virtualizzazione a livello del solo sistema operativo. Similmente alle macchine virtuali, i container forniscono uno spazio isolato e separato per eseguire le applicazioni, pur sfruttando l'hardware condiviso. Il principale fattore di differenziazione è legato al ridotto consumo di risorse, come anticipato in precedenza. Infatti non dovendo inglobare tutte le risorse di un server, i container sono più "leggeri" delle macchine virtuali e possono essere avviati in tempi molto ridotti. Questo li rende particolarmente adatti alle situazioni in cui il carico di lavoro da sostenere è variabile nel tempo e con picchi difficilmente prevedibili. I container quindi limitano il loro livello di astrazione al solo sistema operativo. C'è da aggiungere inoltre che ogni utente oltre a condividere lo stesso sistema operativo, condivide

anche il kernel dello stesso, la connessione di rete e i file di base del sistema. Quindi le istanze vengono eseguite all'interno di uno spazio separato, garantendo così una notevole diminuzione di consumo della CPU e dell'overload associato, che risulta essere un fenomeno tipico dell'esecuzione di più sistemi operativi nelle macchine virtuali. Come per il caso delle macchine virtuali che hanno due tipologie di hypervisor, anche i metodi di gestione dei container si differenziano in due tipologie: Full system Container e Application Container.

1.3.1 Full System Container

Full System Container o più comunemente Operative System Container, condivide il kernel della macchina host, fornendo un isolamento di tipo user namespace [1], come mostrato in figura 1.5. Questo metodo permette alla CPU di partizionare la memoria da allocare in diversi livelli di isolamento. Gli Operative System Container possono essere facilmente comparati agli Hypervisor o alle Virtual Machine in genere. Le applicazioni e le loro librerie possono essere installate allo stesso modo di come avviene in qualunque altro sistema operativo installato su macchina virtuale. Utilizzando i Full System Container, è facile assegnare un indirizzo IP di rete statico o remoto, utilizzare diversi dispositivi di rete, o eseguire comandi di edit sui file in /etc/hosts. Un esempio sono: LXC/LXD, OpenVZ, Oracle Solaris Zone, FreeBSD Jails.

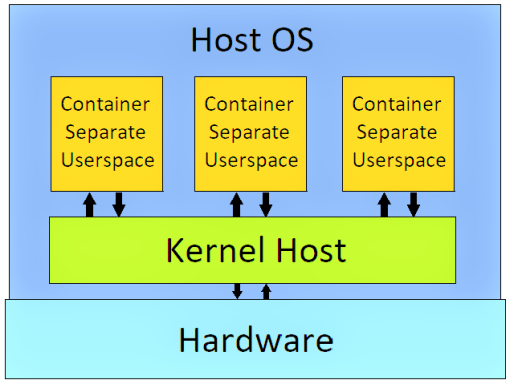

Figura 1.5. OS Container.

1.3.2 Application Container

Application Container allo stesso modo della precedente tipologia, condivide il kernel del sistema operativo della macchina host con il livello soprastante come mostrato nella figura 1.6 che mette in paragone gli application container con i full system container. Questa tipologia di Container è stata progettata allo scopo di poter avviare un singolo processo o applicazione all'interno di ogni container, garantendo un certo grado di isolamento, in modo tale che il processo che andrà in esecuzione all'interno di un container, abbia un proprio filesystem privato e non abbia nessuna visibilità di alcun altro processo che si trova in esecuzione sullo stesso server o macchina host. Questo tipo di isolamento viene ottenuto grazie ai namespaces. Con essi è infatti possibile gestire e isolare le risorse linux quali: *Inter Process Communications*(IPC), configurazione di rete, punto di mount della root, l'albero dei processi, gli utenti e i gruppi di utenti, nonché la risoluzione del nome di rete. Quindi il vantaggio principale sta nel fatto che con l'utlilizzo dei namespaces, diventa possibile isolare i processi in modo efficace, però come detto in precedenza, ogni processo isolato nel container, condivide con il sistema operativo della macchina host, il kernel dello stesso. Al fine di gestire l'accesso alle risorse hardware, i container utilizzano un ulteriore modulo Kernel chiamato Control Groups[1], che ha il compito di impostare le priorità e di misurare diversi tipi di risorse tra cui: la memoria, l'utilizzo della CPU e gli accessi al disco. Un esempio di Application Container sono: Docker e Rocket.

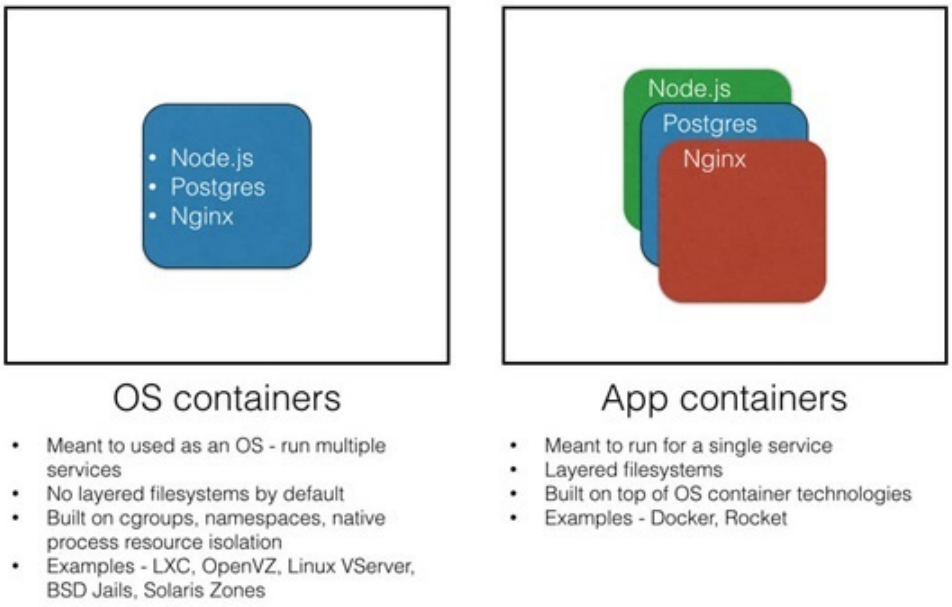

Figura 1.6. Tipologie Container [11].

1.4 Componenti e funzionalità dei Container

In questa sezione ci si focalizzerà sulla descrizione delle varie componenti e funzionalità fornite dal gestore di container.

1.4.1 Container scheduler

Il container scheduler come illustrato in figura 1.7 fornisce la possibilità di definire e attivare un ambiente potenzialmente composto da diversi container. E' quindi un concetto simile a quello dell'orchestrazione negli ambienti cloud basati sulle Virtual Machine. Attraverso il container scheduler si dovrebbe riuscire a definire la corretta sequenza di attivazione dei container, la dimensione del cluster di container che è necessario creare e la qualità del servizio dei container, come ad esempio la richiesta di un minimo garantito di memoria e di accessi al disco da parte di un'applicazione specifica. Tool avanzati per l'orchestrazione, devono invece garantire anche che un determinato ambiente rifletta lo stato desiderato dopo essere stato creato, il che significa che questi strumenti di orchestrazione possono anche svolgere un certo grado di monitoraggio ed effettuare almeno in parte, una autoriparazione nel caso lo stato dell'ambiente non si allinei con quello stabilito dalla configurazione.

Un'altra importante caratteristica comune a molti scheduler per container è la capacità di definire un livello di astrazione intorno a un cluster di container identici, in maniera tale che essi siano visti come servizi, ovvero, un indirizzo IP e una porta. Questo risultato viene ottenuto facendo impostare allo scheduler un load balancer davanti al cluster di container accompagnato dalle regole di indirizzamento necessarie. In quest'ambito, alcuni degli strumenti disponibili sono Kubernetes, Mesos e Docker Compose. Quest'ultimo specifico per Docker.

Kubernets

Kubernetes è un container scheduler reso disponibile in formato open source da Google. Esso, introduce il concetto di pod. I pod sono una serie di container che devono essere collocati insieme nello stesso host. Esistono svariate situazioni o modelli ricorrenti in cui una funzione come questa può essere necessaria, come per esempio: un'applicazione potrebbe aver bisogno di un'altra

applicazione di “accompagnamento” che si occupi di inviare i log. Kubernetes introduce inoltre il concetto di “controller di replicazione” . Il compito del replication controller infatti, consiste nel garantire che tutti i pod di un cluster siano in “salute” . ovvero se un pod è giù o non sta fornendo le prestazioni attese, il replication controller lo rimuoverà dal cluster e creerà un nuovo pod. Oltre a questo, Kubernetes cerca di essere portabile tra implementazioni private di Docker e implementazioni cloud pubbliche, grazie all'uso di una architettura a plugin. Per esempio, la configurazione del load balancer necessaria per la definizione di un servizio, attiverà un plugin differente a seconda che il deploy del progetto sia effettuato in Google Cloud invece che in Amazon Web Service. L'intenzione è quella di aggiungere plugin per collegarsi a diversi tipi di storage, per effettuare diverse tipologie di deployment, per utilizzare fornitori di sicurezza diversi e così via.

Mesos

Mesos è un progetto open source ospitato dalla Apache Foundation. Lo scopo di Mesos sta nel fornire un sistema kernel distribuito. Mesos infati, tenta di trasportare i principi del kernel Linux sul sulla temporizzazione dei processi e applicarli a un cluster o ad un intero datacenter. Il progetto Mesos è cominciato nel 2012 ed è stato recentemente sottoposto a un processo di reingegnerizzazione per renderlo in grado di gestire anche i container Docker. Inoltre, utilizza ZooKeeper per implementare i concetti dei servizi. Mesos ha inoltre delle funzionalità di portabilità che rendono possibile la creazione di cluster Mesos i quali si estendono su diversi datacenter e su differenti fornitori di servizi cloud.

Docker Compose

Docker Compose è un'aggiunta recente all'insieme delle soluzioni Docker. Compose non può essere considerato ancora pronto per la produzione e non è neanche ben integrato con Docker Swarm che è strumento di gestione dei cluster proprietario Docker. Per ora, Docker Compose è in grado di controllare il ciclo di vita di un'applicazione composta di container multipli che sono in esecuzione sullo stesso host.

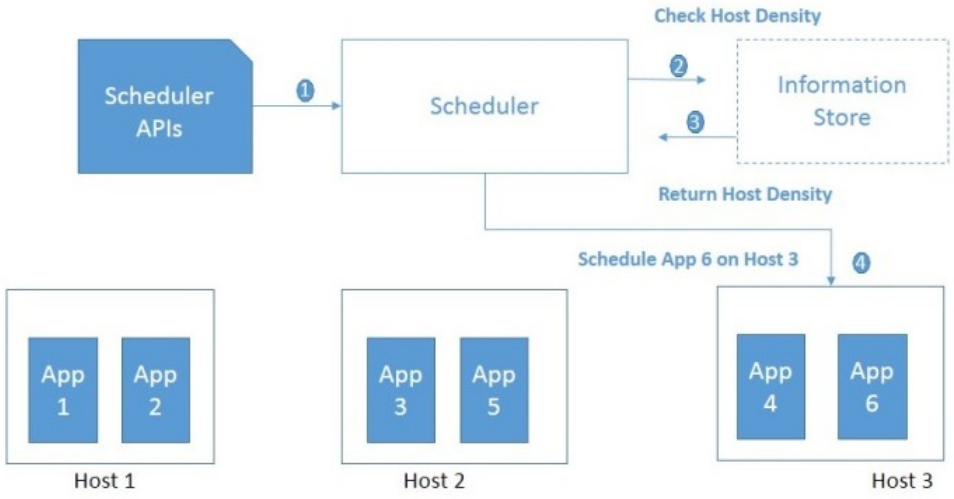

Figura 1.7. Container Scheduler [12].

1.4.2 OCI image

Open Container Iniziative (OCI), come illustrato in figura 1.8, è un progetto Linux Foundation sviluppato con lo scopo di poter utilizzare standard aperti per la virtualizzazione a livello di sistema operativo, soprattutto per i container Linux. Attualmente esistono due specifiche per

lo sviluppo che sono: la specifica di *Runtime Specification (runtime-spec)* e l'*Image Specification* (image-spec). Tali specifiche serviranno quindi a definire la configurazione, l'ambiente di esecuzione e il ciclo di vita di uno specifico container. Ad alto livello, un'applicazione OCI scaricherebbe un'immagine OCI andando ad inserire tale immagine in un pacchetto di file system di runtime OCI. A questo punto il pacchetto di runtime OCI verrebbe gestito da un OCI Runtime. Questo intero flusso di lavoro dovrebbe supportare l'*User Experience*(EX)[2] che gli utenti si aspettano dai motori di container come ad esempio Docker e rkt. In primo luogo infatti, vi è la possibilità di eseguire un'immagine senza ulteriori argomenti, come indicato nell'esempio di seguito.

```
docker run example.com/org/app:v1.0.0

rkt run example.com/org/app,version=v1.0.0
```

Per supportare questo UX, l'OCI image format contiene informazioni sufficienti per avviare l'applicazione sulla piattaforma di destinazione (ad es. Comando, argomenti, variabili d'ambiente, ecc.). Questa specifica definisce quindi come creare un'immagine OCI, che poi verrà generalmente eseguita da un sistema di compilazione, pubblicando un manifest (componente che permette la raccolta delle informazioni basilari di un' applicazione) dell'immagine, una serializzazione dei file system e una configurazione dell'immagine. Ad alto livello, il manifest dell'immagine, conterrà dei metadati relativi al contenuto e alle dipendenze dell'immagine stessa, inclusa l'identità indirizzabile del contenuto di uno o più archivi di serializzazione del filesystem, che verranno poi disimballati per costituire il file system ultimo, quello runnable. La configurazione dell'immagine include inoltre, le informazioni come: gli argomenti dell'applicazione, gli ambienti, ed altro. La combinazione del manifest di un'immagine, con la configurazione dell'immagine e di una o più serializzazioni del filesystem prende quindi il nome di OCI image.

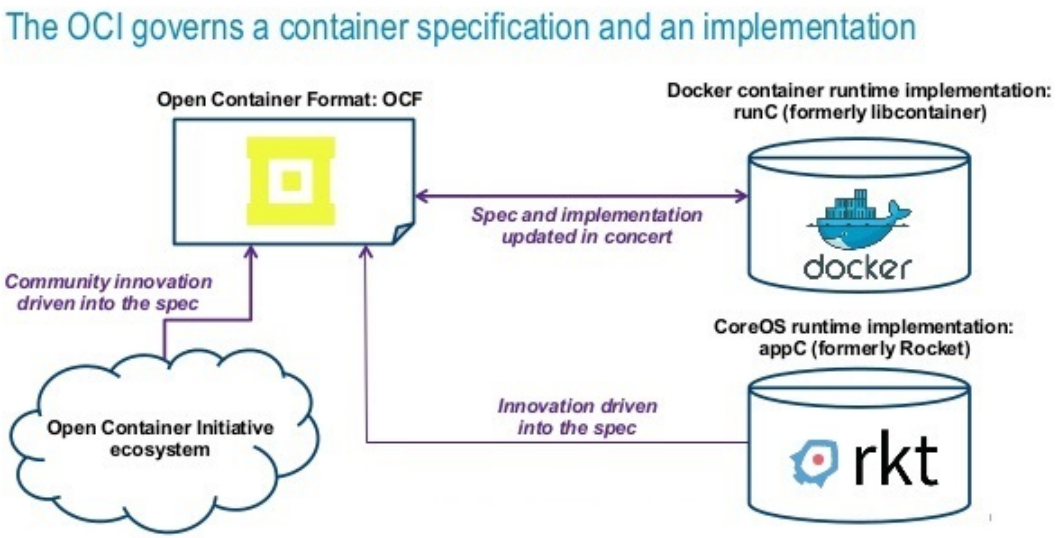

Figura 1.8. Open Container Initiative Image [15].

1.4.3 Checkpoint Restore in Userspace

Checkpoint Restore in Userspace (CRIU) è un meccanismo che consente di salvare lo stato corrente di un processo e di ripristinarlo successivamente al suo stato precedente, ovvero allo stato prima del checkpointing.

Tutte le informazioni relative al processo controllato, sono memorizzate in uno o più file di immagine. Questi file di immagine contengono le informazioni sul processo come ad esempio le pagine di memoria, i descrittori dei file e la comunicazione tra i processi.

È possibile inoltre ripristinare un processo sullo stesso sistema o su un altro sistema differente da quello di origine.

Checkpoint restore è stato creato in *High Performance Computing* (HPC), ovvero tramite tecnologie utilizzate da computer cluster per creare dei sistemi di elaborazione in grado di fornire delle prestazioni molto elevate e nell'ordine dei PetaFLOPS, ricorrendo tipicamente al calcolo parallelo.

Checkpoint restore risulta quindi particolarmente prezioso negli ambienti HPC dove una singola applicazione potrebbe essere distribuita a centinaia o migliaia di core.

In HPC infatti, il fallimento di un singolo componente può portare alla perdita di dati e quindi i cicli della CPU di quelle centinaia o migliaia di nuclei andrebbero per finire sprecati.

Questa tecnologia presenta differenti modalità operative. È infatti possibile utilizzare approcci diversi al fine di evitare la perdita di dati che può esser stata causata da errori di checkpoint e restore.

Ecco alcuni esempi:

- L'applicazione può effettuare il checkpoint e restore di se stessa per memorizzare lo stato corrente.
- L'applicazione può essere controllata e ripristinata in modalità "semitrasparente" intercettando le chiamate di sistema.
- L'applicazione può essere controllata e ripristinata in modalità completamente trasparente a livello di sistema operativo.

Tale strumento viene quindi eseguito a livello di sistema operativo e non richiede alcun prerequisito prima dell'esecuzione del checkpoint e restore. Inoltre non è richiesto l'utilizzo di specifiche librerie per l'esecuzione di tale meccanismo o l'utilizzo di ambienti appositamente preparati al fine di poter intercettare le chiamate di sistema. Tuttavia, questo approccio richiede però l'utilizzo di uno strumento più complesso per il controllo ed il ripristino. Con il crescente interesse per la tecnologia dei container Linux, il checkpoint restore ha iniziato ad attirare più attenzione. Infatti ora è possibile utilizzare il controllo e il ripristino di un processo anche come un mezzo di tolleranza agli errori. E' inoltre possibile utilizzarlo per il bilanciamento del carico, migrando un processo in esecuzione da un sistema all'altro (Live Migration/Cross-Hosting). La migrazione di un processo in esecuzione non è altro che controllare un processo, trasferirlo al sistema di destinazione e ripristinare il processo allo stato originale. La tecnologia Checkpoint e restore ha inoltre la facoltà di poter ripristinare un intero gruppo di processi. Di conseguenza, il checkpoint e restore potrebbe diventare la tecnologia di base ideale per la migrazione dei container. Le prime implementazioni del checkpoint e restore non hanno riguardato l'inclusione a monte. Di conseguenza, nella comunità del kernel Linux non c'era alcun accordo sul progetto. Questo ha portato all'adozione di soluzioni che non sono state ufficialmente accettate dalla comunità Linux. Un'implementazione del checkpoint e restore del kernel è stata sviluppata in collaborazione con la comunità Linux. L'approccio del checkpoint/restore del kernel è diventato troppo complesso per essere integrato nel kernel Linux e pertanto non è stato ulteriormente sviluppato ed è stato abbandonato. Per risolvere i problemi di queste implementazioni precedenti, con CRIU si è deciso di seguire un altro approccio. Infatti ora esegue il maggior numero possibile di funzionalità nello spazio utente e utilizza interfacce esistenti per implementare con successo il checkpoint/restore. Una delle più importanti interfacce del kernel per CRIU è l'interfaccia ptrace. CRIU infatti si basa sulla capacità di cogliere il processo attraverso la ptrace. Quindi, inietta il codice per disporre le pagine di memoria del processo nei file di immagine dall'interno dello spazio di indirizzi del processo. Per ogni parte di controllo del processo vengono creati file di immagine separati. Le informazioni sulle pagine di memoria, ad esempio, vengono raccolte da /proc/PID/smaps, /proc/PID/mapfiles/ e da /proc/PID/pagemap.

I file di immagine delle pagine di memoria richiedono lo spazio di archiviazione, in particolare rispetto ai file immagine rimanenti. I file immagine rimanenti contengono informazioni aggiuntive sul processo di controllo, come ad esempio file aperti, credenziali, registri e stato delle attività.

Per controllare l'albero di un processo (un processo e tutti i suoi processi figlio), i controlli CRIU vengono collegati ad ogni processo figlio.

Per ripristinare un processo, CRIU utilizza le informazioni raccolte durante il controllo. E' importante aggiungere che è possibile ripristinare un processo solo se ha lo stesso *ID processo* (PID) che aveva quando era originariamente checkpointed. Se un altro processo utilizza questo PID, il ripristino non andrà a buon fine.

Una delle ragioni per cui il processo deve essere ripristinato con lo stesso PID è che gli alberi di processo genitore-figlio devono essere ripristinati esattamente come erano in precedenza ovvero allo stato prima del checkpoint.

Non è inoltre possibile ripristinare un processo. Infatti, per ripristinare un processo con lo stesso PID, viene utilizzata un'interfaccia del kernel, che è stata introdotta per influenzare quale PID il kernel fornisce al processo successivo.

Se il processo appena creato con clone() ha il PID corretto, CRIU lo trasforma portandolo allo stesso stato in cui il processo era prima del checkpoint. I file vengono aperti e posizionati come prima, la memoria viene ripristinata allo stesso stato e tutte le altre informazioni rimanenti dai file di immagine vengono utilizzate per ripristinare il processo. Una volta ripristinato lo stato, le rimanenti parti vengono rimosse. Quindi il processo ripristinato riprende il controllo e continua dal punto in cui era precedentemente, ovvero prima del checkpoint, come illustrato in figura 1.9.

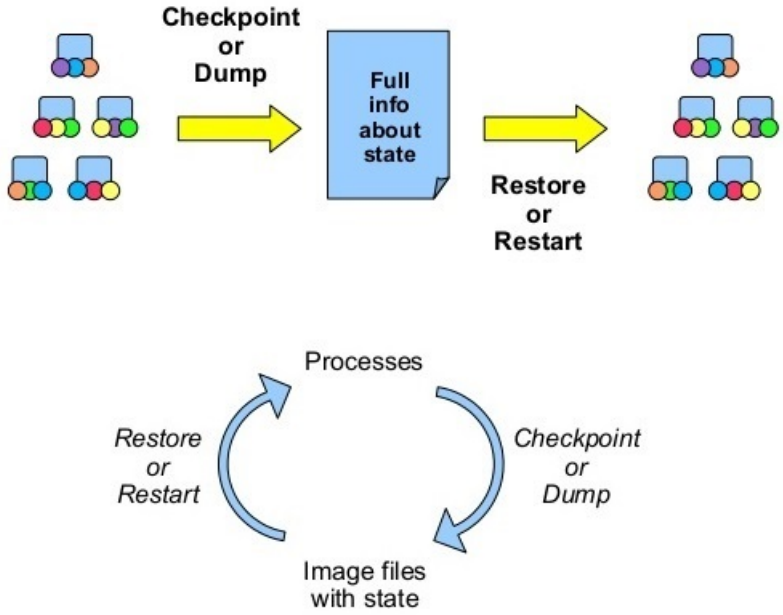

Figura 1.9. Checkpoint Restore in Userspace [17].

1.4.4 Live Migration e Cross-Host

La migrazione in live dei container si riferisce al processo di spostamento dell'applicazione tra macchine fisiche differenti senza dover necessariamente scollegare il client. Una migrazione in diretta può aiutare con scenari di manutenzione del server o nel caso di un carico squilibrato. La memoria, il file system e la connettività di rete dei container in esecuzione sull'hardware, vengono trasferiti dalla macchina host originale ad una nuova macchina host di destinazione, mantenendo lo stato senza interruzioni. La migrazione in live, può essere utile quando ad esempio un amministratore di sistema ha la necessità di effettuare la manutenzione del server o un upgrade dell'hardware, e dovrà quindi spostare i container di tutti gli utenti che sono presenti nella macchina originale in un altro nodo hardware. Un altro scenario di utilizzo si può avere nel caso di carico squilibrato tra

gli host di un cluster, dove uno di essi risulta più carico in termini di utilizzo di risorse rispetto agli altri, richiedendo quindi l'implementazione di modelli applicativi che impongono delle restrizioni sulla scelta dei carichi di lavoro che possono essere ospitati da ogni singola macchina host presente su un cluster . Un ulteriore scenario può consistere nella migrazione dei servizi da un provider di cloud ad un altro nel caso in cui i prezzi e le qualità di tali servizi risultassero troppo costosi.

Ci sono due tipi di soluzioni di migrazione dal vivo. Uno di essi è il pre-copy memory come illustrato in figura 1.10. Se si desidera eseguire la migrazione di un container, la piattaforma tiene traccia della memoria nel nodo di origine e ne fa copia in parallelo con il nodo di destinazione finché la differenza diventa minima. Dopo di che, blocca il container, ottiene il resto dello stato della memoria, la migra nel nodo di destinazione,e in seguito ripristina il nodo destinazione e lo "scongela" per tornare in esecuzione.

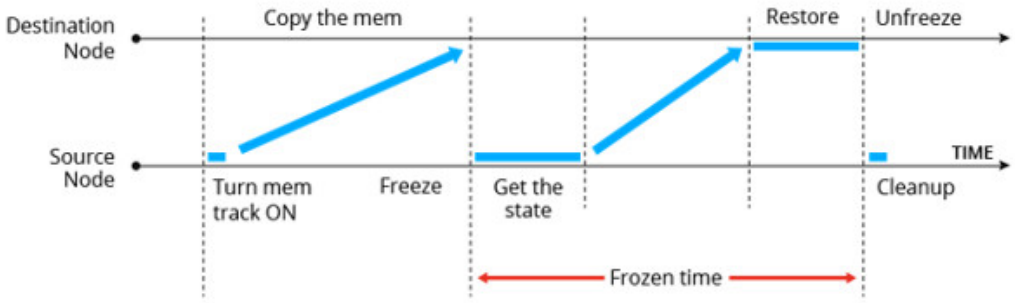

Figura 1.10. Pre-copy memory.

Nella figura 1.10, il source node o nodo di origine, è dove viene inserito un container prima della migrazione in live. Destination node o nodo di destinazione è dove un container verrà posizionato dopo la migrazione in live.

Per eseguire la migrazione, la piattaforma blocca il container al nodo di origine che a sua volta blocca la memoria, i processi, il file system e le connessioni di rete, ottenendo lo stato di questo container. Successivamente, viene copiato nel nodo di destinazione. In seguito la piattaforma ripristina lo stato e riattiva il container nel nodo destinazione. In fine tramite un processo di pulizia rapido, verrà liberato il nodo sorgente.

Quindi è un processo abbastanza diretto, ovvero si ottiene lo stato, si copia lo stato, e si ripristina lo stato. Tuttavia, bisogna notare che esiste un periodo di cosiddetto "congelamento" o frozen time, che è necessario considerare durante lo sviluppo dell'architettura di un' applicazione, perchè per alcune di queste potrebbe essere un problema.

Un'altra soluzione si ha utilizzando il post-copy memory,o la cosiddetta migrazione pigra che è descritta in figura 1.11. Il sistema blocca il container all'inizio del nodo di origine, ne ottiene lo stato delle pagine di memoria che cambiano più velocemente, sposta tale stato nel nodo di destinazione, lo ripristina per poi toglierlo dallo stato di ibernazione nella quale era stato precedentemente posto. Il resto dello stato viene copiato dal nodo di origine alla destinazione in modalità background.

1.4.5 Chroot

Chroot è un metodo che consiste nel cambiare la directory di riferimento dei processi che sono in esecuzione corrente e per i processi generati da questi ultimi (cosiddetti processi figlio) come illustrato in figura 1.12.

Normalmente, le funzioni necessarie a mettere in pratica il chroot sono implementate direttamente nel kernel del sistema operativo. Una di queste è pivot_root, che sposta il file system root del processo corrente alla put_old directory e rende new_root il nuovo file system root, rimuovendo tutte le dipendenze dalla put_old directory.

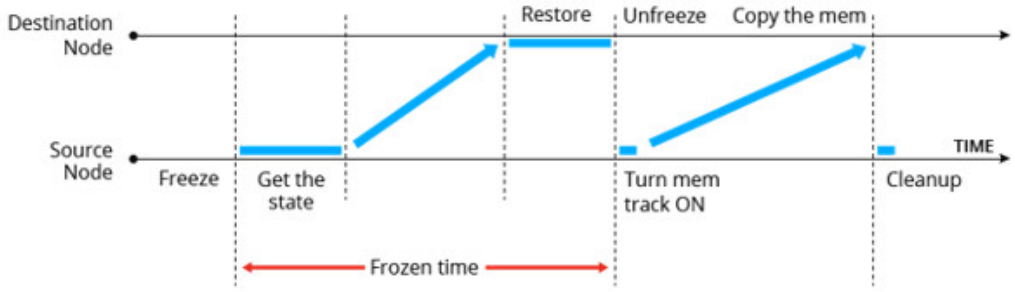

Figura 1.11. Post-copy memory.

Chroot è un metodo che viene utilizzato per isolare i limiti operativi di un'applicazione, ma non è stato concepito come metodo di sicurezza, sebbene alcuni sistemi, come FreeBSD, lo utilizzino come tale ma con l'aggiunta di alcune accortezze. Il nome deriva dal termine informatico root (radice), che indica la directory principale del file system del sistema operativo *Filesystem Hierarchy Standard* (FHS) in cui sono contenute tutte le altre directory.

Normalmente, un software può accedere a tutti i dischi e le risorse del sistema operativo, compatibilmente con i permessi; l'operazione di chroot consiste nell'eseguire il programma bloccato dentro una sottodirectory, permettendogli di accedere solo alle risorse di cui ha strettamente bisogno. La sotto directory in questione viene anch'essa denominata chroot e deve contenere una copia (di solito un hard link) di tutti i file di sistema richiesti dal software. Nel caso di sistemi Unix, è necessaria anche un copia dei device file a cui il programma deve accedere.

Infatti se un utente malevolo riuscisse nell'intento di acquisire il controllo di un'applicazione, avrà tutti i privilegi associati all'applicazione e potrebbe essere in grado di compromettere la sicurezza dell'intero sistema, ad esempio, installando un rootkit. Al contrario, se il programma compromesso si trova in un chroot, dovrà prima uscire dal perimetro e solo in seguito potrà apportare danni al sistema.

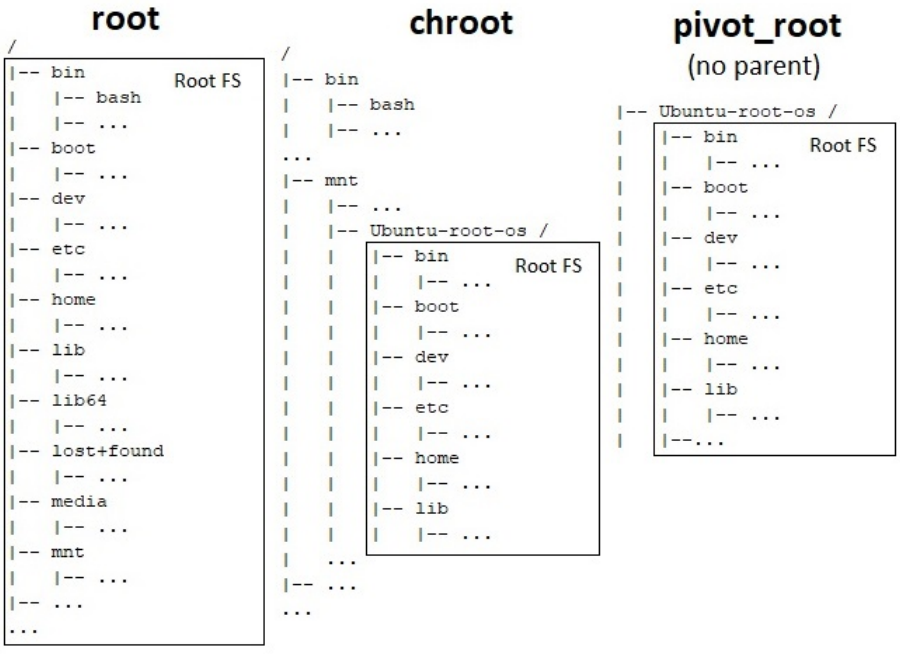

Figura 1.12. Chroot e pivot_root [18].

1.4.6 Networking

Poichè si è riscontrato che ogni rete tende ad avere le proprie esigenze di politica uniche,si è cercato di favorire lo sviluppo di modelli in cui il networking viene disaccoppiato dal container di runtime. Questo migliora notevolmente anche la mobilità delle applicazioni. In questo modello, la rete è gestita da un **plugin** o **driver** che gestisce le interfacce di rete, gestendo inoltre come i container sono connessi alla rete stessa. Il plugin assegna anche l'indirizzo IP alle interfacce di rete dei container. Affinché questo modello possa funzionare, è necessario disporre di un'interfaccia o di un'API ben definita tra il runtime del container ed i plugin di rete. Ci sono due standard per la gestione del networking nei container: Container Network Model e Container Network Interface.

Container Network Model (CNM) come illustrato in figura 1.13 è una specifica di networking di container proposta da Docker. Container Network Model è implementata da Libnetwork, che si occupa di formalizzare i passaggi necessari per implementare una rete tra i container, fornendo un'astrazione che può essere utilizzata per supportare molteplici network drivers.

> "The Container Network Model (CNM) is a specification proposed by Docker, adopted by projects such as libnetwork." [7].

Il CNM è costituito da 3 componenti principali.

Sandbox

Una sandbox contiene la configurazione dello stack di rete di un container. Ciò include la gestione delle interfacce del container, la tabella di routing e le impostazioni DNS. L'implementazione di una sandbox potrebbe essere uno spazio dei nomi di Linux, un FreeBSD Jail o un altro concetto simile. Inoltre una sandbox può contenere molti endpoint di più reti.

Endpoint

Un endpoint unisce una sandbox a una rete. L'implementazione di un endpoint potrebbe essere una coppia di veth, una porta interna vSwitch aperta o simile. Un endpoint può appartenere a una sola rete o anche ad una sola sandbox.

Rete

Una rete è un gruppo di endpoint che sono in grado di comunicare tra loro in modo diretto. L'implementazione di una rete potrebbe essere un Linux bridge, una VLAN, ecc. Le reti consistono in molti endpoint.

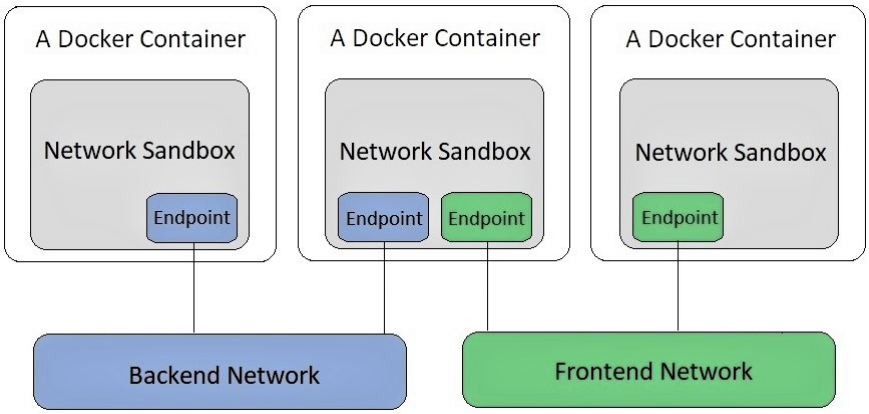

Figura 1.13. Container Network Model. Immagine ispirata da [19]

Container Network Interface (CNI), è un progetto Cloud Native Computing Foundation proposto da CoreOS e consiste di una specifica e di librerie per la scrittura di plugin per configurare le interfacce di rete nei container Linux, insieme ad alcuni plugin supportati.

> "The Container Network Interface (CNI) is a container networking specification proposed by CoreOS and adopted by projects such as Apache Mesos, Cloud Foundry, Kubernetes, Kurma and rkt." [7].

CNI come illustrato in figura 1.14 si occupa di gestire la sola connettività di rete dei container e la rimozione delle risorse assegnate quando il container viene eliminato. Questo repository contiene inoltre, il codice sorgente scritto in linguaggio Go di una libreria per l'integrazione di CNI nelle applicazioni e uno strumento di riga di comando di esempio per l'esecuzione dei plugin CNI. Un repository separato contiene invece i plugin di riferimento e un modello per la creazione di nuovi plugin.

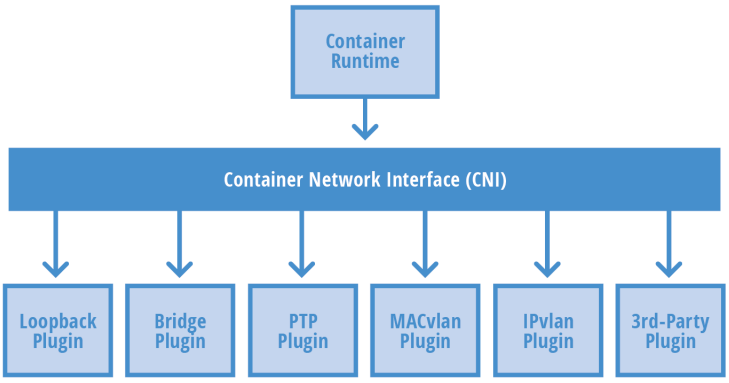

Figura 1.14. Container Network Interface [20].

1.5 Vantaggi nell'utilizzo dei Container

In questa sezione verranno analizzate più nel dettaglio le caratteristiche e i possibili vantaggi dovuti all'utilizzo della tecnologia basata sui container.

Al fine di garantire l'isolamento delle risorse tra il sistema host e i container in esecuzione su di esso, viene implementato un meccanismo del kernel noto come namespace, per fare in modo che ogni processo abbia una propria visione del sistema differente da quella degli altri processi. Le risorse sono quindi effettivamente condivise, ma ogni container vede solo le risorse di filesystem, memoria, processi e dispositivi, come se fossero interamente dedicate ad esso.

Un altro vantaggio è dato dalla possibilità di una migliore gestione delle risorse. Infatti, al fine di gestire l'accesso a tali risorse, i container utilizzano un altro modulo kernel noto come CGroups[1], che permette sia di limitare l'accesso a CPU, memoria e I/O per ciascun container, sia di fare in modo che essi possano accedervi con una maggiore priorità rispetto ad altri.

In seguito i container forniscono uno sviluppo semplificato, ovvero un container impacchetta l'applicazione in una singolo componente che può essere distribuita e configurato con una sola linea di comando e non bisognerà quindi preoccuparsi dell'eventuale configurazione dell'ambiente di esecuzione.

I container permettono inoltre una maggiore disponibilità, ovvero uno sviluppatore ha la possibilità di poter ospitare sul proprio computer uno svariato numero di container. Un container è infatti molto più leggero di una macchina virtuale, la quale quest'ultima può arrivare anche ad occupare diversi Gb. L'esecuzione contemporanea di più macchine virtuali è comunque possibile, ma andrebbe ad impattare pesantemente con le prestazioni del sistema. Cosa che invece non accade con i container, grazie alla loro dimensione ridotta.

I tempi di avvio dei container risultano essere più rapidi rispetto alle macchine virtuali. Infatti, dal momento che si sta virtualizzando solo il sistema operativo e non l'intera macchina, il container

potrà essere avviato in un tempo che, come detto in precedenza, è molto più breve di quello necessario per avviare una virtual machine.

Infine, la portabilità e la consistenza nei container, possono essere probabilmente considerati come i vantaggi principali che questa tecnologia è in grado di offrire, grazie al suo formato che consente l'esecuzione applicativa su host differenti. Inoltre, grazie alla standardizzazione è poi facile spostare velocemente i carichi di lavoro lì dove sono eseguiti in maniera più rapida ed economica, evitando i problemi di compatibilità legati alle particolari caratteristiche delle singole piattaforme dei provider. Il vero punto di forza dei container sta proprio nella minor potenza di calcolo richiesta, nella leggerezza, nella portabilità, nell'isolamento e soprattutto nella capacità di fare a meno dell'hypervisor. Essendo poi basati su tecnologie open-source, questi rappresentano un'ottima soluzione alle richieste di maggiore agilità, risparmio e flessibilità nel contesto delle risorse informatiche che vengono utilizzate per il cloud pubblico,privato o ibrido. I container rappresentano quindi l'elemento perfetto per gestire le complessità, soprattutto perché oltre a possedere una forte componente di automatizzazione dei suoi processi di creazione, avvio e gestione, consentono inoltre anche la riorganizzazione tramite crescenti livelli di astrazione. L'utilizzo di questa tecnologia permette poi di allontanarsi ancora di più dai vincoli hardware permettendo di avere un sistema che presenta una maggiore flessibilità, compattezza e funzionalità sia per le macchina host, sia per i fornitori di servizi e sia per gli utilizzatori finali del sistema stesso.

1.6 Tecnologie per la gestione dei Container

Al crescere del numero dei container in esecuzione su vari host, si è reso fondamentale avere degli strumenti appositi per gestirli in maniera semplice, diretta e con un livello di astrazione anche abbastanza elevato. Tutte le varie soluzioni presenti nel mercato odierno sono di tipo Application container o Full system container, ed operano a differenti livelli di astrazione.

Queste tecnologie sono partite come un'estensione di *Linux Container* (LXC), introducendo una API di alto livello che permette di automatizzare la distribuzione del software in ambiente sicuro e riproducibile, fornendo quindi tutte le funzionalità necessarie per costruire, caricare, scaricare, avviare e arrestare un container. Tali tecnologie sono state pensate per la gestione di processi soprattutto in ambienti single-host che presentano un numero abbastanza piccolo di container.

Quando le applicazioni sono scalate attraverso più host, diventa quindi fondamentale gestire ogni host e astrarne la sua piattaforma sottostante in maniera tale da eliminare eventuali complessità, tramite un meccanismo di orchestrazione. Con il termine orchestrazione infatti, intendiamo proprio la schedulazione dei container, la gestione del cluster e il concetto di approvvigionamento.

La schedulazione si riferisce quindi alla capacità dell'amministratore di caricare su un host un servizio che stabilisce come eseguire un dato container.

Per gestione del cluster invece si vuole intendere il processo di controllo di gruppi di host. Questo può includere l'aggiunta o la rimozione di host dal cluster, l'acquisizione di informazioni sullo stato corrente di host e container, l'avvio e l'interruzione dei processi.

Una delle maggiori responsabilità dello schedulatore è la selezione del'host, infatti se un amministratore decide di eseguire un servizio, ovvero avviare un container all'interno di un cluster, solitamente lo schedulatore selezionerà l'host in modo automatico. L'amministratore può però opzionalmente imporre dei vincoli sulla schedulazione in accordo con i suoi bisogni e desideri, ma alla fine sarà sempre lo schedulatore a decidere se rispettare o meno tali vincoli.

Gli schedulatori più avanzati mettono a disposizione una funzionalità che permette la gestione di gruppi di container allo scopo di poter consentire all'amministratore di trattare un insieme di container come una singola applicazione, mantenendo i vantaggi derivanti dall'uso dei container e riducendo l'overhead addizionale di gestione.

Un concetto legato alla gestione del cluster è quello di approvvigionamento, ovvero quel processo mediante il quale un amministratore di sistema può inserire nuovi host nella rete e configurarli in maniera semplice, così che essi possano essere utilizzati per eseguire determinate operazioni.

Nonostante il risultato dell'approvvigionamento sia sempre la comparsa di un nuovo host online disponibile per eseguire nuove operazioni, la metodologia con cui ciò viene fatto dipende dal tool e dal tipo di host utilizzati. L'approvvigionamento può infatti essere eseguito dall'amministratore, oppure essere inglobato nei tool di gestione del cluster per lo scaling automatico. Questo secondo metodo comporta la definizione di un processo utile per richiedere host addizionali così come la definizione delle condizioni che devono verificarsi affinché questa richiesta venga inoltrata. Ad esempio, se la nostra applicazione soffre di un sovraccarico del server, sarebbe desiderabile che il nostro sistema avviasse nuovi host e scalasse i container attraverso l'infrastruttura al fine di alleviarne la congestione.

Nella sezione successiva descriverò 3 delle tecnologie di container management quali Docker,CoreOS Rkt e LXC, che secondo una pubblicazione della NCC Group "Understanding and Hardening Linux Container" [1], risultano essere le più utilizzate per la creazione e gestione dei container, in aggiunta ad un'estensione di LXC chiamata LXD .

1.7 Docker

Docker nasce come un'estensione delle funzionalità di LXC e viene utilizzato allo scopo di costruire, spostare ed eseguire container Linux based. Docker è un progetto open source nato con un' azienda di supporto che fornisce assistenza e altri servizi a pagamento, il cui punto di forza sta sia nella sua particolare facilità di utilizzo, sia nella possibilità di utilizzare delle API che permettono una virtualizzazione molto leggera, permettendo inoltre di avviare processi isolati fra loro. Docker è stato sviluppato in linguaggio GO ed utilizza LXC, CGroups[1], user namespaces e kernel Linux. Poichè Docker si basa su LXC non include un sistema operativo separato. Docker infatti implementa un'architettura di tipo client-server, dove il client è un semplice strumento a linea di comando che invia comandi al server utilizzando servizi REST. Questo implica che chiunque potrebbe costruire client differenti. Il server invece è un daemone Linux in grado di costruire immagini ed eseguire container sulla base di un'immagine preesistente. Questa tecnologia funziona quindi come un container engine portabile, che permette di inserire un'applicazione e tutte le sue dipendenze in un container virtuale, al fine di poter essere avviato da un qualunque server Linux. I container possono inoltre essere costruiti in maniera interattiva utilizzando un apposito file di configurazione. La costruzione interattiva dei container consente di lanciare alcuni comandi che creano diversi cambiamenti all'interno di un container. Tali modifiche possono essere rese definitive in un secondo momento con un commit verso una nuova immagine di container. C'è da aggiungere inoltre che i container basati su file di configurazione possono essere utilizzati nella costruzione di processi automatizzati. Le immagini Docker possono essere immagazzinate in un repository utile a facilitarne la condivisione. Tale tecnologia di gestione dei container offre inoltre un prodotto chiamato Docker Trusted Registry che è un repository pubblico il quale può essere installato anche in locale.

Per avere una migliore comprensione di Docker, risulta però necessario familiarizzare con le sue componenti principali, le quali sono presentate graficamente in figura 1.15.

Docker Engine

Il Docker Engine è il punto di ingresso principale di Docker. Esso è responsabile della costruzione dell'immagine Docker, dell'orchestrazione, della gestione dei volumi, della creazione di reti e dello scaling.

Docker Command Line

Docker *Command Line*(CL) è l'utilità della riga di comando utilizzata per interagire con Docker.

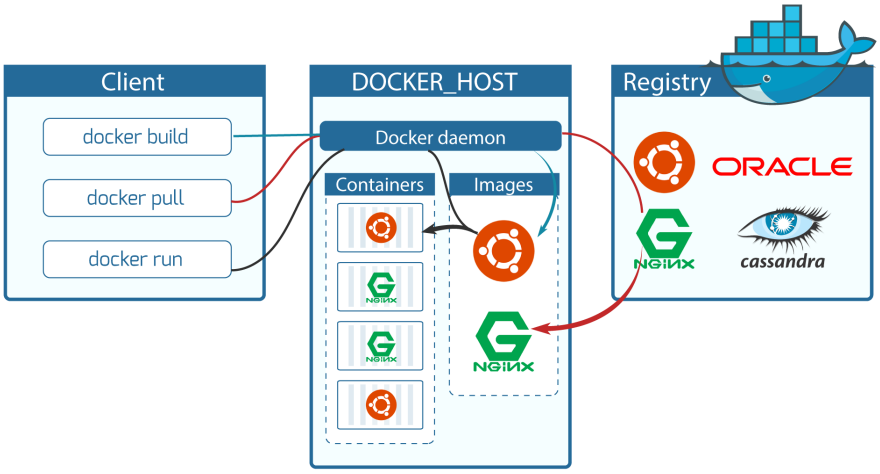

Figura 1.15. Docker Components [21].

Docker Daemon

Una volta che si richiede al Docker Engine di eseguire un'immagine, esso delega la responsabilità al *Docker Daemon* (Containerd). Containerd si occupa di trasferire e memorizzare l'immagine del container, creando il filesystem radice e chiamando il runC tramite containerd-shim, il quale quest'ultimo si occuperà di far si che il runC esca dopo l'avvio del container. In questo modo non è necessario avere i processi di runtime di lunga durata per i container. Tale procedura viene attuata con lo scopo di poter avviare, supervisionare, gestire le basi di storage e le interfacce di rete del container. Al fine di poter instaurare tra il demone Docker e il docker engine, sarà utilizzato un protocollo di comunicazione che prende il nome di gPRC. Containerd resterà quindi in ascolto per richieste attuate dalle Docker API e zi occuperà di gestire i Docker objects come ad esempio le immagini, i container, la rete e i volumi. Il demone può a sua volta comunicare con altri demoni e gestire inoltre i servizi Docker.

Docker client

Il *Docker client* (Docker) è la parte che permette ad uno specifico utente di poter interagire con il Docker stesso. Quando utilizziamo comandi quali ad esempio: docker run, il client invierà tali comandi al dockerd che dopo averli ricevuti si occuperà di eseguirli. Il comando docker utilizza le Docker API, permettendo quindi al Docker client di poter comunicare con più di un demone alla volta.

Docker registry

Il Docker registry è la parte che contiene le immagini Docker. Esempi di esso sono il Docker Hub e Docker Cloud, i quali sono registri pubblici che chiunque può utilizzare per depositare le proprie immagini Docker. Inoltre di default, Docker è configurato per cercare le immagini sul Docker Hub. E' anche possibile avviare, eventualmente, il proprio registro privato. Infatti, se si utilizza *Docker Datacenter* (DDC), questo includerà anche il *Docker Trusted Registry* (DTR). Quando utilizziamo i comandi docker pull o docker run, l'immagine richiesta viene recuperata del registro configurato. E' presente inoltre il Docker store che consente l'acquisto e la vendita di immagini Docker o la distribuzione gratuita. Per esempio, è possibile acquistare un'immagine che contiene un'applicazione o servizio di un software vendor, ed utilizzare tale immagine per il deploy di un'applicazione all'interno del proprio ambiente di testing, staging e produzione. E' inoltre posssibile fare un upgrade di un'applicazione, prendendo la nuova immagine e facendo il redeploing del container.

Continuando con la descrizione delle componenti Docher, durante l'utilizzo dello stesso si vanno a creare ed utilizzare delle immagini, container, reti, volumi, plugin ed altri oggetti che prendono il nome di Docker object. Di seguito farò una panoramica di questi oggetti.

Docker Images

Un' immagine è un template in sola lettura, al cui interno sono contenute le istruzioni per creare il Docker container. Spesso un'immagine si basa su un'altra immagine con l'aggiunta di alcune modifiche. Per esempio, è possibile effettuare il build di un'immagine a partire dall'immagine di base di Ubuntu che installa anche un web server Apache ed altre varie applicazioni, come i dettagli di configurazione utili ad avviare l'applicazione. E' dunque possibile creare la propria immagine o utilizzarne una creata da altri e pubblicata sul repository pubblico. Per fare il build della propria immagine e necessario creare un Dockerfile con una semplice sintassi che definisce i passi necessari alla creazione e all'avvio dell'immagine stessa. Ogni istruzione nel Dockerfile crea un layer all'interno dell'immagine. Quando non modifichiamo o sostituiamo il Dockerfile ed effettuiamo il rebuild, solo i layer che sono stati modificati subiscono il rebuilt. Questo è ciò che rende le immagini così leggere e veloci nell'esecuzione, rispetto alle altre tecnologie di virtualizzazione.

Docker Container

Un container viene descritto come un'istanza di tipo "runnable" di un'immagine. E' infatti possibile creare, avviare, fermare, spostare o cancellare un container, utilizzando le Docker API o CLI. E' inoltre possibile connettere un container ad una o più reti, collegare uno storage ad esso od anche creare una nuova immagine basata sullo stato corrente di tale container. Di default, un container è relativamente ben isolato dagli altri container e dall' host machine. E' anche possibile controllare l'isolamento di rete dei container e degli storage, l'isolamento del sottosistema rispetto ad altri container e rispetto all' host machine. Un container è quindi definito dall'immagine stessa che andrà poi avviata così come le opzioni di configurazione fornite per creare ed avviare tale container. Quando un container viene fermato, ogni cambiamento effettuato in seguito che non sia stato eventualmente salvato viene perso.

1.7.1 Docker e Libcontainer

Libcontainer è una libreria che a partire dalla versione 0.9 si sostituisce ad LXC 1.8, il quale rappresentava l'ambiente di esecuzione precedente. Libcontainer ad oggi rappresenta l'ambiente di esecuzione predefinito di Docker. Questa libreria è stata sviluppata da docker.io, e scritta in linguaggio go e C/C ++, allo scopo di poter supportare un'ampia gamma di tecnologie di isolamento. Con questa libreria si è cercato di andare a creare uno strato di astrazione che fosse in grado di standardizzare il modo in cui viene fatto il packaging, la consegna e l'avvio isolato di applicazioni all'interno di uno specifico container 1.16 1.17. Tale libreria nasce come progetto autonomo, rendendo quindi possibile l'adozione anche a terzi. Ad oggi anche Google, openvz, redhat, ubuntu (lxc) stanno contribuendo a questo progetto. In questo modo, le funzionalità del container disponibili nelle API del kernel di linux vengono fornite in modo coerente grazie all'utilizzo di una libreria univoca. LibContainer affronta inoltre il problema di avere un' unica kernel API che supporta diverse implementazioni come LXC e libvirt [6]. Tale libreria inoltre consente ai container di lavorare con spazi di nomi, gruppi di controllo, capabilities, AppArmor, profili di sicurezza, interfacce di rete e regole di firewall in modo coerente e prevedibile.

> "Libcontainer enables container to work with Linux namespaces, control groups, capabilities, AppArmor, security profiles, network interfaces and firewalling rules in a consistent and predictable way. It includes namespaces, standard filesystem setup, a default Linux capability set, and information about resource reservations. It also has information about any populated environment settings for the processes running inside a container." [13].

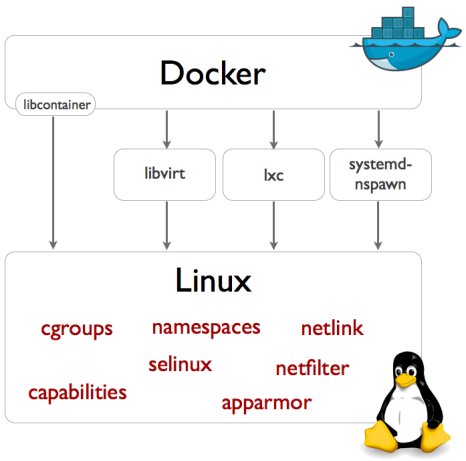

Figura 1.16. Libcontainer [13].

Attualmente, container management system come Docker possono supportare direttamente queste funzionalità del kernel, in quanto come detto in precedenza, Docker con l'avvento di libcontainer non dipende più da LXC. Infatti grazie al fatto che non ci si basi più sulle componenti degli spazi Linux, si è riuscito a ridurre drasticamente il numero di parti in movimento e ad isolare il container Engine da problemi che erano stati inizialmente introdotti nelle versioni e nelle distribuzioni che adottavano LXC linux container, anzichè libcontainer.

> "Libcontainer drastically reduces the number of moving parts, and insulates Docker from the side-effects introduced across versions and distributions of LXC. In fact, libcontainer delivered such a boost to stability that we decided to make it the default." [5].

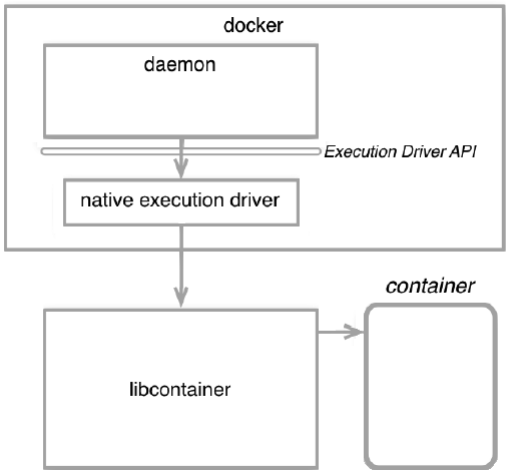

Figura 1.17. libcontainer. [14]

1.7.2 Docker e RunC

RunC è un tool che si occupa di mandare in esecuzione un container. Inizialmente container engine come Docker, funzionavano mediante l'utilizzo diretto degli strumenti di *Linux container* (LXC), per gestire i container su piattaforma Linux. Successivamente si sono spostati su **libcontainer**, una libreria che permette di interfacciarsi direttamente con le feature del kernel necessarie all'esecuzione dei container, come namespaces[1] e cgroups[1]. Quindi runC è uno strumento che

viene gestito da riga di comando e permette l'esecuzione di un container tramite libcontainer, indipendentemente dall'engine che si sta utilizzando, prendendo un container conforme OCI 1.4.2 e mandandolo in esecuzione.

Abbiamo inoltre un altro componente che collabora con runC ed è **containerd**. Containerd è un demone che utilizza runC (o una qualsiasi alternativa, purché conforme OCI) allo scopo di poter gestire i container ed esporre le sue funzionalità tramite gRPC.

> "Containerd can manage the complete container lifecycle of its host system: image transfer and storage, container execution and supervision, low-level storage and network attachments, etc.. Containerd includes a daemon exposing gRPC API over a local UNIX socket. The API is a low-level one designed for higher layers to wrap and extend. It also includes a barebone CLI (ctr) designed specifically for development and debugging purpose. It uses runC to run container according to the OCI specification." [3]

In pratica, fornisce un'interfaccia *Create Read Update Delete* (CRUD) per la gestione specifica dei container.

> "When we are building APIs, we want our models to provide four basic types of functionality. The model must be able to Create, Read, Update, and Delete resources. Computer scientists often refer to these functions by the acronym CRUD. A model should have the ability to perform at most these four functions in order to be complete. If an action cannot be described by one of these four operations, then it should potentially be a model of its own.
>
> The CRUD paradigm is common in constructing web applications, because it provides a memorable framework for reminding developers of how to construct full, usable models." [4]

Infatti a differenza del Docker Engine che permette di gestire anche immagini, volumi, network, etc. Una descrizione grafica di RunC è illustrata in figura 1.18.

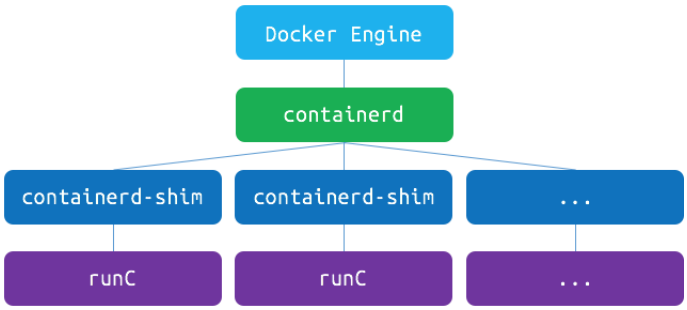

Figura 1.18. Runc e Containerd [16].

Docker Servicies

Un servizio permette ad un container di passare attraverso dei Docker daemons multipli, che collaborano tra loro con lo scopo di formare uno swarm di manager and workers. Ogni membro di questo swarm è un Docker daemon e questi membri comunicano fra loro attraverso le Docker API. Un servizio dunque, permette di definire lo stato desiderato come un numero di repliche del servizio che deve essere disponibile ad ogni istante di tempo. Di default un servizio è load-balanced attraverso tutti i worker nodes. Al consumer, il servizio Docker appare come un'applicazione singola. Il Docker engine supporta però la funzionalità di swarm mode a partire dalla versione 1.12. Nella figura 1.19, viene illustrata l'architettura Docker a partire dalla versione v1.11. Infatti, fino alla versione 0.8, Docker utilizzava LXC per interagire con il kernel Linux. Dalla versione 0.9 fino alla 1.10, Docker interagisce direttamente con il kernel Linux utilizzando l'interfaccia libcontainer, che in seguito subirà un repackaging nella versione 1.11.

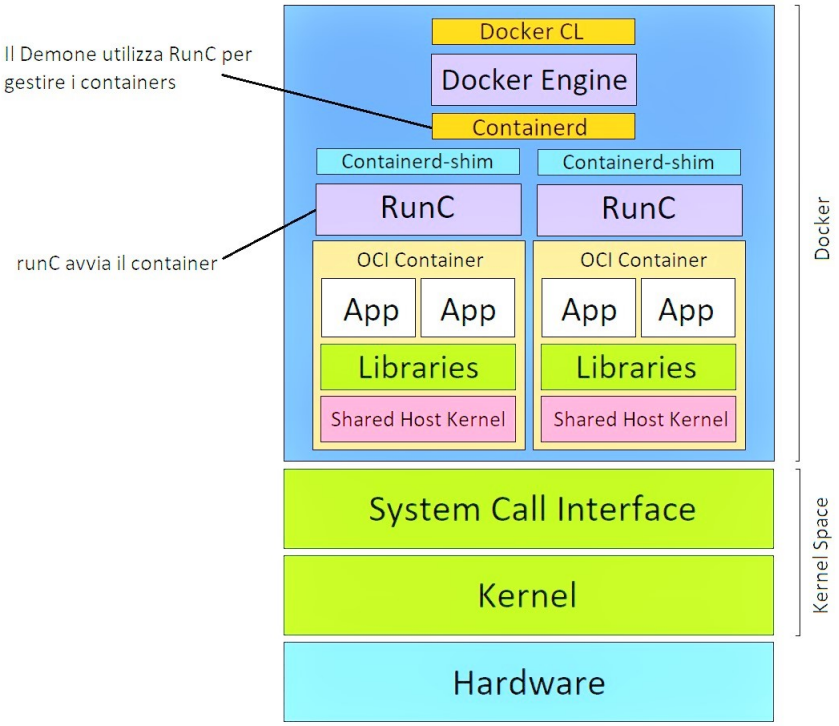

Figura 1.19. Docker v.1.11 Architecture [43].

1.8 LXC Linux Container

Linux container rappresenta una tecnologia o ambiente di virtualizzazione che opera a livello di sistema operativo e che consente di eseguire differenti processi Linux ciascuno in un proprio ambiente isolato (container) su una macchina host avente un kernel Linux, come rappresentato in figura 1.20. Un processo in esecuzione all'interno di un container avrà quindi un suo file system privato e quindi non avrà visibilità di alcun altro processo che andrà in esecuzione sullo stesso server Linux. Questo tipo di isolamento viene ottenuto grazie ai namespaces. Con i namespaces infatti, è possibile proteggere le risorse del kernel linux quali: l'*Inter Process Communications* (IPC), la configurazione di rete, il punto di mount della root, l'albero dei processi, gli utenti e i gruppi di essi, nonché la risoluzione del nome di rete.Quindi il vantaggio principale sta nel fatto che, con l'uso dei namespaces, diventa possibile isolare i processi in modo più efficace.

L'unica cosa che il processo isolato nel container condivide con il sistema operativo ospitante è il kernel Linux e quindi anche parte delle risorse utilizzate dai container vengono condivise tra essi, a meno di opportuni meccanismi di isolamento delle risorse stesse tra i vari container.

Bisogna tenere in considerazione che nativamente il kernel Linux fornisce già due meccanismi per gestire l'isoloamento delle risorse hardware e dei namespeces (o cosiddetto spazio dei nomi), quali CGroups[1] e user namespaces[1]. La prima funzionalità permette di gestire le priorità e di limitare l'utilizzo di risorse di CPU,memoria, accesso alla memoria, accessi ai dischi, rete, senza dover necessariamente utilizzare una macchina virtuale. La seconda funzionalità permette invece di isolare tra di loro i processi o gruppi di processi, come alberi di processi, risorse di rete, user ID e mounted file system. Questo tipo di virtualizzazione OS-level, risulta simile ad altri progetti presenti in Linux quali ad esempio: OpenVZ e Linux-VServer. In altri sistemi operativi Unix invece, si hanno delle somiglianze con le workload partitions AIX, le jail di FreeBSD e i container di Solaris.

LXC non è quindi un sistema di virtualizzazione vero e proprio che si occupa di emulare l'hardware basandosi su un Hypervisor, ed inoltre, supporta la virtualizzazione dei soli sistemi

Linux. I container fanno eseguire direttamente le istruzioni alla CPU senza la necessità di adottare meccanismi di emulazione o di compilazione in tempo reale, portando quindi ad un utilizzo di risorse e un overhead molto basso rispetto alla virtualizzazione classica basata su virtual machine.

Fino alla versione di Linux kernel v3.8 un utente con privilegi di root poteva eseguire codice a suo piacimento direttamente sul sistema host, godendo dei privilegi massimi. Con le nuove versioni del kernel e di LXC, ai container viene posto il limite di accesso alle risorse hardware.

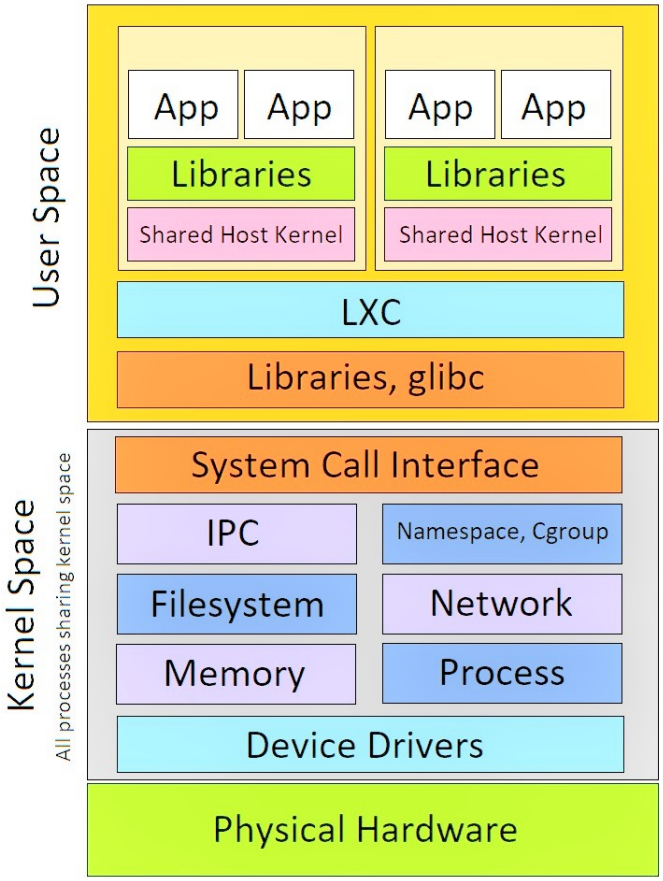

Figura 1.20. LXC Architecture [44].

1.9 LXD Linux Daemon

LXD daemon, come rappresentato in figura 1.21 è una tecnologia di tipo full system container, dove al suo interno può essere avviato un sistema linux completo allo stesso modo di come viene fatto in una macchina fisica o in una virtual machine (dove viene virtualizzato l'intero hardware). Questa tecnologia è costituita da un demone che fa uso delle REST API per gestire i container, in modo da fornire un'esperienza utente simile alle macchine virtuali ma senza il bisogno di virtualizzare l'hardware.

LXD si basa su liblxc ed ha capacità aggiuntive rispetto a lxc come le funzionalità di snapshot e live migration. Inoltre utilizza lxc tramite liblxc e Go language, per la creazione e la gestione dei container, risultando quindi linkato ad lxc.

I container sono isolati tramite CGroups[1] che ne limita le risorse hardware e User Namespaces[1] che viene utilizzato per definire uno spazi dei nomi e decidere quali risorse i processi in esecuzione in tali namespaces possono vedere, senza dover necessariamente ricorrere ad una macchina virtuale.

LXD utilizza inoltre le REST API per gestire i container e quindi per interfacciarsi con liblxc, al fine di poter utilizzare le capabilities fornite da liblxc,e gestire i container, in modo tale che il demone LXD non sia il "central point of failure" e quindi, che i container possano continuare la loro esecuzione anche nel caso in cui il demone LXD debba essere riavviato.

Inoltre, il demone LXD fornisce funzionalità di hypervisor allo stesso modo delle macchine virtuali. Di seguito descriverò le componenti principali di LXD.

Container

I container in lxd sono composti da un filesystem (rootfs) ed un insieme di opzioni di configurazione, incluse le *resource-limit*, *environment* e le *security-option*.

Snapshot

Sono identici ai container eccetto per il fatto che sono immutabili, ovvero possono essere rinominati,distrutti o ripristinati, ma non possono essere modificati. Questo è utile, perchè permette il rollback del container, conservando lo stato di cpu e memoria al momento dello snapshot. La funzionalità che si occupa dello snapshot è CRIU 1.4.3.

Images

LXD è image based, infatti tutti i container vengono generati da un'immagine. Tipicamente si fa riferimento ad immagini "pulite" di varie distribuzioni linux, come avviene per le Virtual Machine. E' inoltre possibile pubblicare un container facendone un'immagine che potrà essere utilizzata in qualunque Host LXD remoto o locale. Ogni immagine è identificata unicamente da una chiave di hash di tipo SHA256.

Profiles

I profili, sono dei meccanismi che permettono di definire delle configurazioni e dei dispositivi per i container una sola volta, per poi essere applicati in seguito ad un vasto numero di container. Inoltre ad un container LXD possono essere applicati diversi profili, e all'atto dell'avvio della configurazione finale anche detta "expanded configuration" , verrà caricato il profilo in base all'ordine con il quale questi sono stati precedentemente definiti, andando a sovrascrivere la configurazione precedente, nel caso in cui essa fosse presente. Ci sono 2 tipi di configurazioni di base, ovvero quella di default e la configurazione docker. La configurazione di default, viene applicata automaticamente a qualsiasi container a meno che non siano presenti profili generati dall'utente. Tale profilo infatti, si occupa solo di definire a "eth0" il dispositivo di rete del container. Il profilo docker ivece, è applicabile a container nel quale si vuole avviare docker. Questo richiede che LXD avvii alcuni moduli kernel necessari per poter abilitare l'annidamento dei container o il cosiddetto "container nesting" .

Remotes

LXD è il demone remoto, ovvero, è uno strumento che tramite command-line lato client permette di interfacciarsi o di comunicare con molteplici server LXD o con le immagini dei server stessi. E' inoltre possibile avviare un certo numero di hosts LXD, configurati per restare in ascolto sulla rete locale.

Controllo avanzato delle risorse

Le risorse quali CPU, memoria, I/O di rete, blocco I/O, utilizzo disco e risorse kernel, come anche le funzionalità di sicurezza sono fornite e gestite da CGroups[1] e User Namespaces[1] attraverso l'uso della libreria LXC (liblxc) insieme al Linux Security Module AppArmor e Seccomp.

Il kernel namespace, a differenza di LXC viene utilizzato di default, e fornisce all'utente l'accesso alla gestione del container, senza privilegi, a meno che non sia strettamente necessario.

Seccomp e le sue funzionalità sono implementate anch'esse di default, al fine poter di filtrare system call potenzialmente pericolose.

AppArmor è anch'esso implementato di default, e fornisce restrizioni addizionali sui mounts, socket, ptrace e sull' accesso ai file. Inoltre limita di default la comunicazione tra i container (cross-container communication).

Le capabilities invece, vengono utilizzate per evitare che i container possano avviare determinati moduli kernel, o che possano modificare il timer di sistema tramite la chiamata della syscall SYS_TIME.

CGroups, viene utilizzato per limitare le risorse hardware di ciascun container, limitandone così possibili attacchi di denial of service. La comunicazione tramite i container LXD avviene tramite TLS 1.2. Quando invece dobbiamo iniziare una comunicazione con un host che si trova al di fuori della *system certificate autority* (SCA), LXD si dovrà occupare di validare la firma (SSH style) dell' host remoto e di memorizzarla in cache per usi futuri.

Rest api

Come anticipato in precedenza, la REST API è il canale di comunicazione tra il client e il demone LXD. Il canale per la REST API può essere instaurato tramite socket linux, oppure tramite socket HTTPs, utilizzando il certificato del client per effettuare l'autenticazione. Nel caso in cui fosse necessario un meccanismo di comunicazione più complesso, come ad esempio la migrazione di container, LXD si occuperà della negoziazione di web-sockets.

Container scaling

LXD permette inoltre la gestione di container multipli e la possibilità di scalarli orizzontalmente sull'hardware di host multipli interconnessi tramite interfacce di rete, permettendo quindi ai container di comunicare tra loro tramite meccanismi di inter process communication, grazie all'utilizzo di un plugin di OpenStack (nova-lxd).

Unprivileged container

Ogni container viene avviato di default senza privilegi di amministratore. Infatti, risultano presenti dei privilegi di root in ogni container ma sono mappati allo stato di utente normale e non di amministratore. Le Network API sono disabilitate di default, permettendo ai container di rimanere in ascolto nella sola rete locale. Inoltre nativamente non viene fornito alcun network management tool, e storage management tool.

Cross-host

LXD fornisce anche il supporto per la migrazione in live dei container ed il trasferimento di immagini, con la possibilità di spostare un container da un host ad un altro senza dover necessariamente arrestare il container stesso.

1.10 CoreOS Rocket

Prima di descrivere le caratteristiche e funzionalità di Rocket, descriverò lo strato di sistema operativo, sul quale Rocket si appoggia di default. CoreOS lo strato che permette di mandare in esecuzione il gestore di container Rocket, è una distribuzione Linux che è stata costruita allo scopo di rendere eseguibili grandi e scalabili implementazioni su un'infrastruttura variegata da

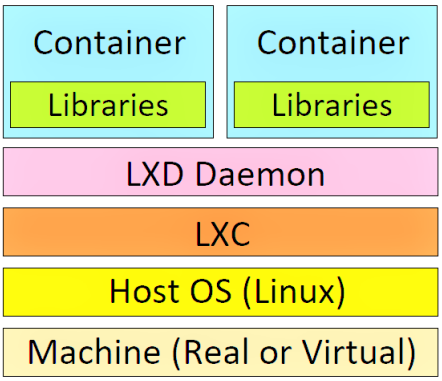

Figura 1.21. LXD Engine.

gestire. Sulla base di una creazione di Chrome OS, CoreOS mantiene un sistema host leggero e utilizza i container Rocket per tutte le applicazioni. Questo sistema fornisce inoltre, l'isolamento del processo consentendo anche di spostare facilmente applicazioni in cluster differenti da quello di origine.

Per poter gestire questi cluster, CoreOS utilizza un archivio di valore distribuito globalmente e denominato etcd, che serve essenzialmente per passare i dati di configurazione tra i vari nodi. Questa componente è anche la piattaforma che permette il discovering dei servizi e consente di configurare dinamicamente le applicazioni in base alle informazioni disponibili attraverso la risorsa condivisa.

Per pianificare e gestire applicazioni nell'intera fascia, viene invece utilizzato uno strumento chiamato fleet. Fleet infatti, funge da un sistema di init di tipo cluster che può essere utilizzato per poter gestire i processi su tutto il cluster. Tutto ciò rende più facile la configurazione delle applicazioni, permettendo inoltre di poter gestire il cluster da un solo punto. Questo viene fatto collegando il sistema di init di ogni singolo nodo.

Di seguito verranno descritte le caratteristiche di Rocket.

Esso infatti è un progetto che ha avuto inizio quando Docker presentava diversi problemi di sicurezza. Rocket era stato infatti disegnato per essere più sicuro e open container rispetto a Docker.

> "Prior to version 1.8, Docker didn't have a way to verify the authenticity of a server image. But in v1.8, a new feature called Docker Content Trust was introduced to automatically sign and verify the signature of a publisher.
>
> In rkt, signature verification is done by default. So, as soon as a server image is downloaded, it is cross checked with the signature of the publisher to see if it is tampered in any way. " [23].

Un ulteriore vantaggio di Rocket in ambito di sicurezza dei container, rispetto alle versioni precedenti di Docker, è dovuto all'esecuzione di questi senza alcun privilegio di root associato ai processi in esecuzione all'interno dei container, a differenza di Docker che di default, eseguiva i container con privilegi di amministratore.

> "Docker runs with super-user privileges (aka "root"), and spins off new containers as its sub-process. The issue with that is, a vulnerability in a container, or poor containment can give an attacker root level access to the whole server. CVE-2014-9357 was one such vulnerability.
>
> Rkt came up with a better solution where new containers are never created from a root privileged process. In this way, even if a container break-out happens, the attacker cannot get root privileges. " [23].

Rkt, come mostrato in figura 1.22 è una tecnologia di tipo Application Container come Docker ed è quindi basata sulla condivisione del kernel del sistema operativo della macchina host tra i vari container.

Tale tecnologia permette l'avvio di un'applicazione o processo all'interno di ogni container assieme alle relative dipendenze, o condividendo queste ultime quando necessario tra le applicazioni dei vari container. Questa tecnologia di container è stata sviluppata come detto in precedenza per l'avvio di un singolo processo all'interno di ogni container, allo scopo di garantirne un certo livello di astrazione rispetto alla macchina sottostante ed agli altri container presenti sulla macchina host. In questo modo ogni processo in esecuzione avrà il proprio file system privato e non potrà vedere alcun altro processo in esecuzione su altri container.

Questo tipo di isolamento è garantito grazie all'user namespace[1], che permette di isolare le risorse linux quali IPC, la configurazione della rete, l'albero dei processi, utenti e gruppi di utenti, la risoluzione del nome di rete, ed infine il punto di mount della root. Con l'utilizzo dei namespaces [1] si ha quindi il vantaggio di isolare i processi efficacemente ma senza la posibilità di isolarli completamente, poiché come detto in precedenza il kernel del sistema operativo presente nella macchina host è condiviso tra i vari processi in esecuzione su ciascun container. Per la gestione delle risorse hardware invece, Rkt utilizza allo stesso modo di Docker (poiché entrambi si basano su linux Container) un ulteriore modulo Kernel chiamato CGroups, con il compito di gestire e limitare le risorse hardware.

Ogni processo creato all'interno di un nuovo container non ha permessi di super-user. In questo modo si garantisce che anche se un container venisse infettato in alcun modo, l'utente malevolo avrebbe accesso al solo container e non all'intera macchina host.

Rkt fornisce inoltre la possibilità di scaricare immagini server ottimizzate per specifiche applicazioni da dei registri pubblici e in tutta sicurezza, poiché effettua la verifica della firma (signature verification) di default, in modo da poter appurare che l'immagine che verrà scaricata non sia stata manomessa in alcun modo.

La pubblicazione delle immagini e condivisione con altri partners tecnologici risulta più semplice e sicura poiché Rocket utilizza il protocollo HTTPS per il download delle immagini, usufruendo inoltre di meta dati memorizzati sul webserver che ha l'immagine memorizzata, allo scopo di poter puntare correttamente alla sua locazione.

HTTPS è una delle possibili soluzioni per la distribuzione di immagini. Infatti utilizzando il comando rkt torrent pull è possibile scaricare le immagini anche via torrent.

Rkt infine fornisce una buona portabilità delle sue immagini in altri sistemi container, poichè utilizza un formato opensource conosciuto con il nome di "appc" , in modo tale che ogni immagine creata da Rocket possa essere portata in un qualunque altro sistema di gestione container che però supporti il formato "appc" .

Con l'utilizzo di questo formato opensource, Rkt non vincola alcun vendor ed aiuta ad una migrazione più "indolore" , nel caso in cui un altro gestore di cotainer che supporti il formato "appc" abbia dei requisiti migliori.

In figura 1.22, il nodo CoreOS esegue il servizio etcd, systemd e i servizi fleet in tutti i nodi del cluster. Gli etcd, che sono in esecuzione in tutti i nodi, si parlano tra di loro ed eleggono un nodo come nodo master. Tutti i servizi in esecuzione all'interno del nodo verranno pubblicizzati in questo nodo master, che rende etcd un meccanismo di rilevamento dei servizi. Allo stesso modo, fleetd che sarà in esecuzione in diversi nodi, si occuperà di mantenere l'elenco dei servizi in esecuzione sui vari nodi presenti nel suo pool di servizi, allo scopo di poter fornire un buon livello di orchestrazione a livello di servizio. Fleetctl e etcdctl sono dunque delle utility di riga di comando che servono per configurare rispettivamente le utilità fleet e etcd. Queste componenti messe insieme, forniscono tre funzionalità di base per CoreOS quali il Service discovery, il Cluster management ed infine il Container management (Rocket e/o Docker).

Service Discovery

Poiché le applicazioni e gli ambienti di rete variano notevolmente tra le implementazioni dei client, fleet non fornisce una soluzione generalizzata e integrata per la scoperta dei servizi. Tuttavia,

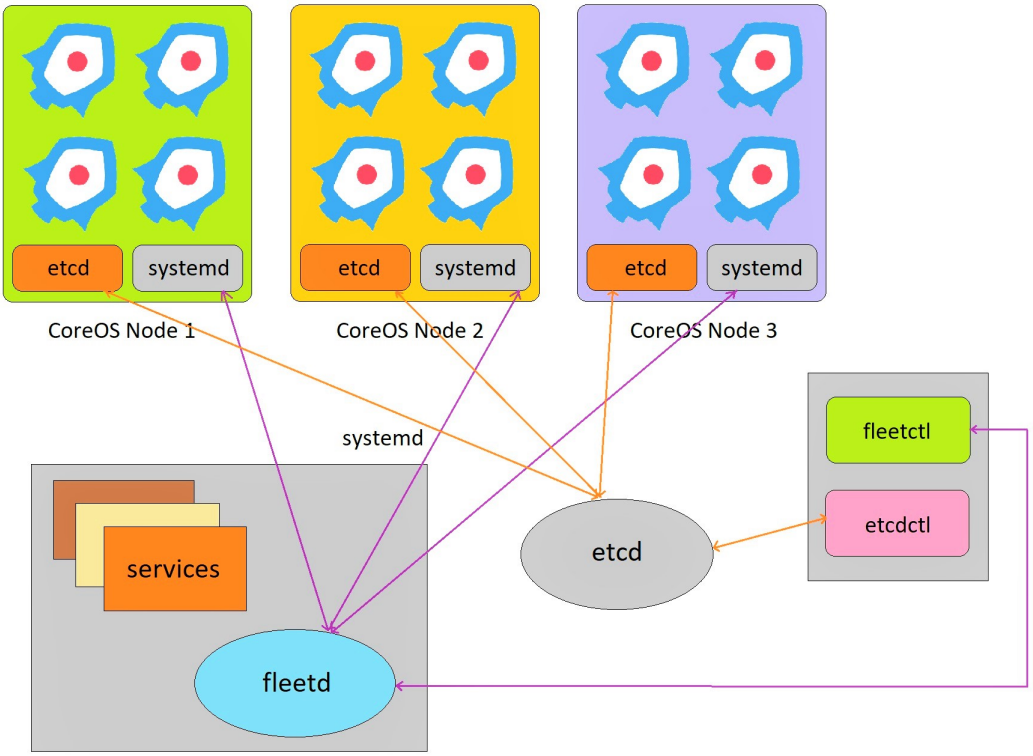

Figura 1.22. CoreOS Rocket Architecture. Immagine ispirata da [22].

esistono diversi modelli che possono essere facilmente implementati in cima a fleet allo scopo di poter fornire un service discovery automatico e affidabile. Uno di questi modelli, è il modello sidekick.

Il modello sidekick infatti, gestisce un agente di discovery separato e posto accanto al container principale che verrà eseguito. Ciò può essere facilmente realizzato in fleet con l'opzione MachineOf. Quindi, invece di indovinare quando un'applicazione è sana e pronta a servire il traffico, è possibile scrivere un agent che ad esempio controlli l'application endpoint /v1/health dopo il deployment. Per un'altra applicazione, si potrebbe desiderare di annunciare ogni istanza da un indirizzo IP pubblico invece di un IP privato.

Cluster management

Un gestore di cluster è solitamente un'interfaccia grafica di back-end (GUI) o un software di riga di comando che viene eseguito su uno o su tutti i nodi di un cluster (in alcuni casi viene eseguito su un server o su differenti cluster di server di gestione). Il cluster management infatti, funziona insieme ad un cluster management agent. Questi agents vengono eseguiti su ciascun nodo del cluster allo scopo di poter gestire e configurare i servizi, una serie di servizi o gestire e configurare il server cluster completo. In alcuni casi, il gestore di cluster viene utilizzato per lo più per la spedizione dei lavori cluster (o cloud) da eseguire. In questo ultimo caso un sottoinsieme del cluster management potrà essere un'applicazione remota che non viene utilizzata per la configurazione, ma solo per inviare lavoro e per ottenere i risultati di lavoro dal cluster stesso. In altri casi il cluster è più correlato alla disponibilità e al bilanciamento del carico rispetto ai cluster di servizi computazionali o specifici.

1.11 Confronto tra le 4 tecnologie di gestione dei container

Nella tabella 1.1 ho elencato le funzionalità e le componenti delle quattro tecnologie di gestione container che ho menzionato e descritto nelle sezioni precedenti, allo scopo di poterle confrontare e capire quale di queste possa essere la più completa in termini di caratteristiche è di funzionalità.

L'elenco indicato nella tabella 1.1 presenta funzionalità come la tipologia dei container management system 1.3, l'interattività con tali tecnologie 1.9, il supporto al cross-hosting 1.4.4, i meccanismi di ripristino di uno specifico container 1.4.3, il supporto alle varie tecnologie di gestione delle reti all'interno dei container, la scalabilità dei container da una a più macchine fisiche, il formato delle immagini supportato da ciascuna tecnologia, le varie tipologie di filesystem supportati ed il tipo di crittografia delle informazioni in transito tra i vari container attraverso le reti locali o remote.

Container management features				
	docker	rkt	LXC	LXD
Tipo di container management	Application Container	Application Container	Full System Container	Full System Container
Interattività	REST API	REST API	REST API	REST API
Supporto Cross-host	supportato	supportato	supportato	supportato
Checkpoint restore in userspace	Default	Default	Default	Default
Gestione avanzata di rete	Container network Model (CNM) 1.4.6	Container network interface(CNI) 1.4.6	bridging, NAT, static IPS, public IPs, IPSEC, VPNs	bridging, NAT, static IPS, public IPs, IPSEC, VPNs
Scalability	distribuzione di un singolo container su più macchine fisiche	distribuzione di un singolo container su più macchine fisiche	distribuzione di un singolo container su più macchine fisiche	distribuzione di un singolo container su più macchine fisiche
Formato immagini supportate	Docker,OCI 1.4.2	OCI, Application container images (ACI)	Unified Tarballs,Split Tarballs	Unified image (single Tarball),Split image (two Tarballs)
Tipo di crittografia	TLS	PGP,TLS over HTTP	TLS	TLS
Filesystem supportati	Overlayfs, Overlayfs2, Aufs, Btrfs, ZFS e plugins (Flocker, Convoy)	Overlayfs, Aufs	Btrfs, LVM, ZFS	Btrfs, LVM, ZFS

Tabella 1.1. Container management features

Capitolo 2

Problemi di sicurezza noti in ambito container

In questo capitolo, ci si focalizzerà sulla descrizione dei vari problemi di sicurezza noti in ambito container, iniziando però con l'introduzione e descrizione dei limiti di sicurezza, che ad oggi sono presenti nei container.

2.1 Sicurezza e limiti dei container

I Container al contrario delle macchine virtuali dispongono attualmente dello stesso kernel della macchina host. Questo risulta essere un grande problema, poiché un errore o bug sul kernel stesso potrebbe consentire ad un processo o applicazione in esecuzione all'interno di un container di acquisire il controllo completo dell'intera macchina sottostante e degli altri container presenti in essa. Quindi i produttori di tecnologie per la gestione dei container si stanno preoccupando di trovare soluzioni che permettano di limitare la superficie di attacco da parte di un utente malevolo e stanno cercando di adottare misure di sicurezza native per controllare ciò che i processi all'interno dei container sono in grado di fare. Inizialmente tecnologie di gestione dei container come Docker aggiungevano tutte le funzionalità a un singolo file di programma monolitico, il che significava che il numero di linee di codice continuava a crescere ogni qual volta veniva rilasciata una nuova versione. Ed è proprio in questo punto che risiede un importante problema di sicurezza. Infatti una singola vulnerabilità in una delle migliaia di righe di codice può portare a compromettere l'intero programma e consentire l'accesso completo ad un utente malintenzionato.

Soluzioni come Rkt, al contrario utilizzano un'architettura modulare in cui le funzionalità vengono sviluppate come file binari indipendenti e da fornitori differenti. Il codice base è mantenuto il più piccolo possibile allo scopo di poter ridurre la superficie di attacco. Questo assicura che anche se una sotto componente venisse compromessa, il danno sarebbe limitato a quello che il programma può accedere. Inoltre è molto importante che gli amministratori dei sistemi container applichino le misure di sicurezza imparate nel corso degli anni e che vengono adottate sui sistemi Linux e Unix, senza dover fare completo affidamento sulle misure di sicurezza implementate nativamente all'interno dei container management, perché molto spesso non sono sufficienti.

Bisogna quindi tenere in considerazione i seguenti accorgimenti:

- Eseguire solo immagini che hanno dato esito positivo alla verifica dell'autenticità stessa e quindi utilizzare strumenti che permettano di firmare immagini e di verificare l'autenticità della firma stessa.
- Le applicazioni all'interno dei container dovrebbero essere necessariamente avviate senza alcun privilegio di root, in modo da evitare che un bug in un'applicazione possa fornire il completo accesso alla macchina sottostante.

- Assicurarsi che il kernel del sistema operativo sia sempre aggiornato, poiché risulta essere condiviso tra tutti i container avviati sulla macchina host.
- Sono necessari strumenti di supporto per ispezionare ed identificare possibili vulnerabilità nel kernel.
- Non devono mai essere disabilitate le features di sicurezza presenti nel sistema operativo.
- Esaminare le immagini del container allo scopo di identificare difetti di sicurezza se presenti ed assicurarsi che il provider li corregga tempestivamente.

2.2 Vulnerabilità note in ambito container

L'avvento delle tecnologie di gestione dei container, in alternativa all'utilizzo delle macchine virtuali ha portato grandi vantaggi in materia di utilizzo delle risorse del sistema, poichè i container risultano essere più leggeri rispetto alle macchine virtuali, dato che non necessitano di un sistema operativo avviato al loro interno per eseguire le applicazioni. Questo però oltre a fornire sistemi più performanti, porta lo svantaggio di avere un kernel condiviso tra i vari container, data l'assenza del sistema operativo guest, e per questo modivo, aumenta la superfice di attacco e quindi anche le vulnerabilità in ambito container. Di seguito descriverò le principali vulnerabilità che ad oggi sono presenti in ambito container.

Compromissione del kernel

A differenza di una Virtual Machine, il kernel viene condiviso tra tutti i container e con l'host stesso, ampliando l'importanza di eventuali vulnerabilità presenti nel kernel. Se un attaccante compromette il sistema host, l'isolamento dei container e le garanzie di sicurezza non saranno più garantite. In una Virtual machine invece la situazione risulta essere differente, infatti un attaccante dovrebbe portare a buon fine un attacco sia sul kernel della macchina virtuale, che sull'hypervisor prima di poter toccare il kernel della macchina host. Un opzione di best practices per la protezione dell' host linux si basa sull'assicurarsi che la configurazione dell'host e della tecnologia di gestione dei container sia sicura, ovvero garantendo un accesso limitato e autenticato alle risorse (senza dover necessariamente fornire i permessi di root massimi) e crittografando la comunicazione. E' inoltre importante utilizzare strumenti di controllo per verificare di aver applicato le migliori pratiche di configurazione.

Un altro aspetto da tenere in considerazione è quello di mantenere il sistema sempre aggiornato. Inoltre, utilizzando sistemi host "minimi" , come ad esempio CoreOS, Red Hat Atomic o RancherOS, si andrà a ridurre la superficie di attacco permettendo inoltre l'esecuzione dei servizi di sistema nei container. È molto importante applicare anche il controllo di accesso obbligatorio allo scopo di poter impedire operazioni indesiderate sia sull'host che sui container e a livello di kernel, utilizzando strumenti come Seccomp, AppArmor o SELinux.

Container breakout

Con il termine di "breakout" si vuole intendere che un dato container ha superato i controlli di isolamento acquisendo sia privilegi aggiuntivi, sia l'accesso alle informazioni sensibili dell'host. Per evitare questo, è necessario ridurre i privilegi di default di un container. Ad esempio, se il demone di un container engine, venisse eseguito di default con privilegi di root, questo permetterebbe di creare uno spazio dei nomi a livello utente e di eliminare o aggiungere alcune delle funzionalità root del container, andando quindi a concedere la possibilità ad un utente malintenzionato che fosse riuscito ad acquisire l'accesso ad uno specifico container, di accedere ad altri container o all'host stesso. Generalmente per impostazione predefinita, gli utenti non sono separati da namespaces e quindi un qualsiasi processo che esce dal container avrà gli stessi privilegi anche nella macchina host sottostante. Quindi se un'applicazione in avvio su un container ha i privilegi di root, questi privilegi si avranno anche nella macchina host. Ciò significa che bisogna

preoccuparsi di potenziali privilage-escalation attacks, per cui un utente guadagna privilegi elevati, come quelli dell'utente root, spesso attraverso un bug nel codice applicativo che deve essere eseguito con privilegi aggiuntivi. Un buona prassi è quella di ridurre le funzionalità (capabilities) solo a quelle necessariamente richieste dal software in secuzione sul container. Ad esempio, la capability CAP_SYS_ADMIN abilitata all'interno di un container è particolarmente pericolosa in termini di sicurezza, poichè garantisce una vasta gamma di autorizzazioni di livello root, come ad esempio il montaggio di file system, l'inserimento di spazi dei nomi del kernel e operazioni di controllo degli input/output dei dispositivi (ioctl). Un'altra buona prassi è quella di creare uno user namespace isolato per limitare i privilegi massimi dei container sull'host all'equivalente di un utente normale, ed evitare l'esecuzione di container come uid0, quando possibile. Se invece fosse necessario eseguire un container con privilegi di root, bisogna verificare che l'immagine avviata in tale container provenga da fonte certa e attendibile. Anche l'accesso al filesystem dovrebbe essere impostato con i privilegi di sola lettura e non lettura/scrittura, in modo tale da non dare accesso in scrittura senza saperne il motivo.

Image Integrity

E' possibile che immagini che si desidera avviare sul proprio server possono essere manomesse, introducendo del codice malevolo che permette all'utente malintenzionato di prendere il controllo completo del sistema host sottostante. E' quindi molto importante esaminare le immagini, prima di scaricarle e mandarle in esecuzione, allo scopo di identificare difetti di sicurezza, se presenti, ed assicurarsi che essi vengano corretti tempestivamente del provider. E' quindi importante utilizzare meccanismi di verifica della provenienza e dell'autenticità di un'immagine che è presente su un repository, vereficare quali sono le politiche di sicurezza utilizzate in queste immagini e avere prova oggettiva crittografica che l'autore di quell'immagine sia realmente chi afferma di essere. Quindi è necessario assicurarsi dell'autenticità di un'immagine prima che essa venga scaricata ed eseguita nella macchina host, per evitare la compromissione dell'intero sistema. Soluzioni basate su *Public Key Infrastructure* (PKI) chain of trust (ovvero la cosiddetta catena di fiducia) permettono di verificarne autenticità,integrità, non riproduzione e controllo di accesso ai dati digitali.

Compromissione di segreti

Quando un container accede a un database o ad un servizio, probabilmente questo richiederà un segreto, come ad esempio una API key o uno username e password. Un attaccante che può accedere a questo segreto avrà anche accesso all' intero servizio. Questo problema diventa più acuto in un'architettura basata su microservices, in cui i container stanno in un continuo start and stop rispetto ad un'architettura con un piccolo numero di virtual machine a lungo termine.

Resource Starvation

Mediamente i container in esecuzione su un sistema, risultano essere molto più numerosi rispetto alle macchine virtuali, ed oltre ad essere più leggeri, possono anche essere facilmente raggruppati in cluster e mandati in esecuzione su hardware modesti. Questo è sicuramente un vantaggio, ma che a sua volta implica che molte entità software sono in competizione per accedere alle risorse della macchina host sottostante. La presenza di bug nel software, errori di calcolo di progettazione o un attacco da parte di un utente malevolo che ha inserito del malware, possono facilmente causare un denial of service a meno che non vengano configurati correttamente i limiti delle risorse messe a disposizione di ogni applicazione in esecuzione su ciascun container. Per ovviare a tale problema, è quindi necessario gestire adeguatamente le risorse di CPU, di memoria RAM, memoria fisica, risorse di rete, risorse di I/O e di swapping, oltre ad alcune risorse kernel ed alla gestione degli spazi utente. Generalmente i limiti di queste risorse sono disabilitati per impostazione predefinita nella maggior parte dei sistemi di gestione dei container. Quindi un corretto utilizzo dei meccanismi di isolamento delle risorse come cgroups permette di evitare possibili attacchi di Denial of Service, limitando le risorse hardware per ciascun container a quelle di cui necessariamente necessita, in modo tale che in caso di attacco vengano esaurite solo le risorse dedicate al container in questione e non le risorse dell'interno sistema host. Come detto in precedenza, oltre all'isolamento delle

risorse sono necessari meccanismi per l'isolamento dei container in spazi utente dedicati e ben distinti fra loro, come user namespace. Infatti, nei container non tutte le risorse alla quale questo può fare accesso sono namepaced. Le risorse che sono namedpaced vengono mappate ad un valore separato nell'host, come ad esempio se consideriamo un processo PID 1 all'interno di un container, questo non sarà identificato come PID 1 nell'host o in qualsiasi altro container. Al contrario, tutte le risorse che non dispongono di un namespace, risulteranno essere le stesse sia nell'host che negli altri container.

Gestione del materiale crittografico

Generalmente può capitare che un'applicazione all'interno di un container utilizzi uno strumento per la gestione e la conservazione di chiavi crittografiche (Kernel keyring). Bisogna quindi essere consapevoli che le chiavi crittografiche sono distinte da uno spazio utente (UID), il che significa che qualsiasi utente con lo stesso UID potrà avere accesso alle stesse chiavi. Quindi il kernel stesso e tutti i suoi moduli (come nel caso in cui un container carichi un modulo del kernel con priviligi di root), saranno disponibili in tutti i container e nell'host stesso. Ciò include moduli come: dispositivi inclusi dischi, schede audio, GPU e l'orologio di sistema (cambiando l'orario all'interno del container si andrebbe a modificare anche l'ora del sistema host sottostante). Questo è possibile solo per i container che hanno la possibilità di effettuare la chiamata di sistema SYS_TIME, che non viene mai assegnata di default.

Linux Kernel exploit

Fondamentalmente, questa vulnerabilità è basata sulla generazione di race conditions nel kernel. La race condition si attua tra due operazioni. La prima operazione va ad applicare il *Copy On Write* (COW) nella memoria mappata in sola lettura, mentre la seconda operazione continua a disporre di quella memoria mappata in sola lettura. Quando questa operazione si andrà ad attuare per un certo numero di volte, si verificherà una race condition, dove il kernel potrebbe andare accidentalmente a scrivere dati nella memoria mappata in sola lettura, anzichè eseguirne una copia privata per la scrittura, abilitando quindi un processo a scrivere dati o informazioni in una sezione di memoria che dovrebbe essere protetta. Per poter eseguire l'exploit,è necessario quindi che il processo acceda alla sua zona di memoria. Questo può esser fatto chiamando ptrace (che richiede la capacità SYS_PTRACE) o aprendo la propria memoria come un "file " tramite /proc/self/mem. Ma poichè l'approccio SYS_PTRACE è meno utilizzabile nei container (questa funzionalità non viene aggiunta ai container per impostazione di default), ci si focalizzerà sui POC di /proc/self/mem per vedere se presentano minacce in ambienti di container. Apparentemente, tutti i POC disponibili ottengono privilegi di root scrivendo a file o oggetti di memoria in modo tale da causare l'escalation privilege solo all'interno del container. Tuttavia, ciò può essere fuorviante, infatti vedremo il perchè. Inizialmente ho supposto di voler scrivere i dati in un file protetto da scrittura, eseguendo inoltre un POC che scrive i dati in un supporto di sola lettura dall'interno di un container. Quindi, premesso che un host può condividere file e directory con un container con restrizioni specifiche come la sola lettura, se un container risultasse in grado di scrivere dati in file di sola lettura, a quel punto si sarebbe verificato il breakout del container, con conseguente possibilità di modificare i dati sensibili o i file che permetteranno di ottenere un innalzamento dei privilegi all'interno dell'host, anche se un utente con privilegi di root non avesse l'accesso in scrittura ad un volume di sola lettura che è mappato in un container (per non parlare di un utente non root). Se invece si avviasse il container in questione in uno user namespace senza alcun privilegio di root, l'utente che sta eseguendo il container verrà mappato su un UID non radicato nell'host, garantendo quindi che nel caso di un breakout del container l'utente sarà effettivamente un utente senza alcun privilegio di root anche nell'host, ma se si volesse riutilizzare l'exploit per andare a modificare un file in sola lettura, anche in questo caso, questo verrebbe modificato. Questo perchè trattandosi di una vulnerabilità nel kernel e questo stesso kernel è condiviso tra i container in esecuzione sull'host, tale exploit potrebbe arrecare dei danni ovunque all'interno del sistema.

Host Rebinding Attack

Un nuovo tipo di attacco portato a Docker versione per Windows, consente agli avversari di abusare dell'API per nascondere codice malevolo su sistemi mirati, permettendo inoltre l'esecuzione di codice remoto. L'idea è stata sviluppata dai ricercatori di Aqua Security e tale tecnica è stata dimostrata alcuni mesi fa alla Black Hat da Sagie Dulce, ricercatore di sicurezza senior, assieme alla Aqua Security. L'attacco funziona in qualunque installazione di Docker che espone la sua API tramite TCP, che fino a poco tempo fa era l'impostazione predefinita per i PC con sistema operativo Windows che eseguono Docker. L'endgame di attacco si basa sull'esecuzione di codice remoto persistente all'interno di una rete aziendale, e la persistenza di tale malware sui computer host risulta praticamente impossibile da identificare dai vari meccanismi di sicurezza presenti nelle macchine host. L'attacco risulta quindi essere di tipo multistadio, dove la prima fase si basa sull'attrarre lo sviluppatore che sta eseguendo Docker per versioni Windows, in una pagina web gestita da utenti malevoli che ospiterà al suo interno del codice javascript specifico per questo tipo di attacco. Inoltre il codice javascript risulterà essere in grado di bypassare la Same Origin Policy security del browser,(una funzionalità implementata nei browser moderni per la protezione dei dati) consentendo inoltre un solo subsett di metodi http, tra cui GET, HEAD e POST. Durante la dimostrazione di tale attacco è stato generato un container Docker sulla macchina host, che utilizza un repository Git per il comando ed il controllo e tutto ciò senza violare la protezione *Same Origin Policy* (SOP). In questa dimostrazione il container presentava funzionalità limitate, ma quello che realmente si voleva fare era acquisire l'accesso all'intera Docker API, in modo tale da poter eseguire qualunque container, come ad esempio un container con privilegi di root e possibilità di accedere all'host o alla macchina virtuale. Per attuare questa procedura, i ricercatori dell'Aqua, hanno creato una tecnica di Host Rebinding Attack, che si basa su un attacco già esistente chiamato DNS Rebinding Attack come mostrato in figura 2.1, che permette ad un utente malevolo di "abusare" del DNS per poter ingannare il browser e bypassare la SOP security. Host Rebinding Attack mette a punto dei protocolli di risoluzione dei nomi di Microsoft, per raggiungere lo stesso scopo, questa volta però servendosi di un'interfaccia virtuale, garantendo a questo attacco la possibilità di non essere rilevato sulla rete, come descritto dall'articolo pubblicato dalla Aqua Security "How Abusing Docker API Led to Remote Code Execution, Same Origin Bypass and Persistence in The Hypervisor via Shadow Container" [9].

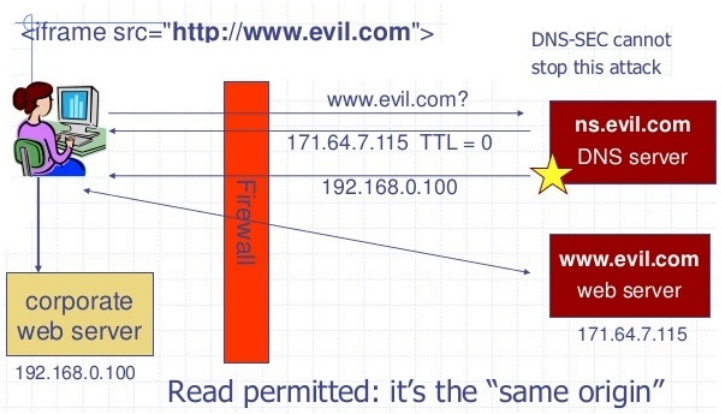

Figura 2.1. DNS Rebinding Attack [24].

2.3 Recenti vulnerabilità che hanno portato ad attacchi in ambito container

Per ogni architettura, oltre ai diversi tipi di attacchi generici elencati e descritti di cui sopra, sono presenti dei CVE noti, ed in particolar modo verranno introdotti quelli con uno score superiore a 4. La tabella 2.1 rappresenterà invece, i CVE noti distinti per architettura e che sono stati

pubblicati dall'anno 2016 ad oggi. Per la lista completa invece si potrà fare riferimento al sito "CVE - Common Vulnerabilities and Exposures (CVE) " [48].

CVE-2017-1000364 (Stack Clash Vulnerability)

La vulnerabilità chiamata "Stack Clash" è stata recentemente scoperta nei sistemi basati su Linux e risulta essere un altro problema di sicurezza critico come il Dirty Cow. Tale vulnerabilità (CVE-2017-1000364) infatti, influenza anche i container. Un utente malintenzionato potrebbe sfruttare la vulnerabilità per ottenere privilegi di root all'interno di un container, ma non necessariamente essere in grado di uscire dal container. Tuttavia, se l'exploit si verificasse nello spazio utente dell'host, permetterebbe all'attaccante di compromettere i container in esecuzione sulla macchina host. Questa vulnerabilità, come detto in precedenza, è stata scoperta di recente dai ricercatori Qualys (un vendor di sicurezza), ed è stata chiamata "Stack Clash " perché coinvolge regioni di memoria come lo stack e l'heap allo scopo di "farli scontrare" . Ciò potrebbe causare il danneggiamento della memoria di un processo dedicata allo stack, o alle regioni di memoria adiacenti, andando anche a causare un possibile aumento dei privilegi del sistema. Qualys ha dimostrato come Stack guard-pages, una componente di sicurezza che era stata progettata appositamente per prevenire questo tipo di problema, sia stata facilmente aggirata da Stack Clash. Le distribuzioni Linux interessate a questo tipo di vulnerabilità, sono: Red Hat, Debian, Ubuntu, SUSE e CentOS, di cui la maggior parte di tali distribuzioni hanno già rilasciato delle correzioni. Mentre questo risulta essere un exploit locale dedicato allo spazio user, vi è anche la possibilità che questo tipo di attacco fornisca un foothold all'interno del sistema operativo o del kernel stesso. Stack Clash potrebbe essere quindi utilizzato anche con altri exploit allo scopo di poter causare ancora più danni sui sistemi. Ad esempio, una recente vulnerabilità sudo (CVE-2017-100367) potrebbe essere combinata con Stack Clash per poter eseguire codice arbitrario con privilegi di root.

Sebbene Stack Clash sia un exploit che generalmente "limita" il suo attacco allo spazio utente e quindi un container dovrebbe confinare tale problema, questo tipo di vulnerabilità può risultare ancora pericolosa. Infatti se le impostazioni di sicurezza dei container come user namespaces e cgroups sono state configurate correttamente, l'utente malintenzionato anche se riuscisse ad avere la meglio in questo tipo di attacco, sarebbe comunque limitato al container stesso. Ma questo però non è comunque detto. Infatti un utente malintenzionato potrebbe combinare Stack Clash con un' altra vulnerabilità, al fine di riuscire ad attraversare il container o anche la macchina virtuale, partendo dal cosiddetto "security kill chain" . Alcuni accorgimenti che potrebbero essere utilizzati per proteggere i container e gli hosts da questi tipi di exploit, potrebbero essere quelli di eseguire una scansione in real-time, monitorare container e host per identificare problemi di privilege escalations e adottare meccanismi di break out detection.

CVE-2016-8649 (lxc-attach vulnerability)

Un recente attacco portato ai gestori di container e più precisamente ad LXC si basa su una vulnerabilità nota (CVE-2016-8649) presente in lxc-attach, comando che permette di avviare un processo all'interno di un container in esecuzione. Infatti nelle versioni precedenti alla 1.0.9 e 2.x prima di 2.0.6 lxc-attach consente ad un utente malintenzionato che riesce a portare a buon fine un attacco ad un container avviato senza alcun privilegio di root, di utilizzare un descrittore di file ereditato da /proc dell'host per accedere al resto del filesystem dell'host tramite la famiglia di syscall openat(). Tale attacco comporta una totale divulgazione delle informazioni presenti nell'host, in quanto tutti i file di sistema vengono rivelati. Inoltre vi è una totale compromissione dell'integrità del sistema, con conseguente e totale perdita della protezione del sistema. L'utente malintenzionato può inoltre rendere completamente inaccessibili le risorse interessate per tale attacco. Per portare a buon fine un tale attacco non è quindi necessario avere grandi conoscenze o abilità nel campo della sicurezza. La vulnerabilità sulla quale si basa questo attacco richiede inoltre che un utente malintenzionato sia connesso al sistema, ad esempio tramite interfaccia a riga di comando, tramite una sessione desktop o un' interfaccia web.

CVE-2016-1582

LXD prima della versione 2.0.2 non impostava correttamente le autorizzazioni per la commutazione delle autorizzazioni di un container dalla modalità non privilegiata a quella privilegiata, e questo consentiva agli utenti della rete locale di accedere a percorsi arbitrari in sola lettura nella directory del container tramite vettori non specificati. Questo portava ad una notevole divulgazione delle informazioni, ma senza apportare alcun impatto sull'integrità del sistema stesso. Inoltre per portare a segno tale attacco non sono necessarie competenze specifiche e non è richiesta l'autenticazione per sfruttare tale vulnerabilità.

CVE-2016-1581

LXD prima di 2.0.2 utilizza permessi di lettura per /var/lib/lxd/zfs.img quando imposta un pool ZFS basato su loop, che consente agli utenti locali di copiare e leggere dati da un qualsiasi container tramite vettori non specificati. Anche in questo caso la divulgazione delle informazioni può risultare notevole, ma senza alcun impatto sull'integrità del sistema. La complessità di tale attacco non è elevata, quindi non sono richieste particolari competenze in ambito della sicurezza e non è richiesto alcun meccanismo di autenticazione per sfruttare tale vulnerabilità.

CVE-2016-6595 (Docker resource starvation vulnerability)

Una recente vulnerabilità al toolkit SwarmKit 1.12.0 per Docker ha consentito agli utenti malevoli autenticati da remoto, di portare a segno un attacco di denial of service tramite una lunga sequenza di join e quit actions. Il Docker swarmkit infatti viene utilizzato per il coordinamento delle macchine o nodi in un swarm. Una volta che una macchina si unisce diventa un nodo Swarm. I nodi possono quindi essere distinti in worker nodes o in manager nodes. Tale vulnerabilità consiste nel fatto che una nuova macchina non riesce a fare il join nel cluster swarm, dopo che un'altra macchina ha eseguito il join e il quitting dal cluster swarm per un certo numero di volte, mandando in blocco il dispatcher e causando l'uscita del server della *Certificate Authority* (CA). Tale vulnerabilità non ha alcun impatto sulla riservatezza delle informazioni e sull'integrità del sistema. Vi è però una elevata riduzione delle prestazioni con conseguente interruzione della disponibilità delle risorse hardware. Non sono necessarie competenze in ambito della sicurezza per portare a segno tale attacco, ma tale vulnerabilità richiede che un utente malevolo sia connesso al sistema, ad esempio mediante riga di comando, sessione desktop o tramite un' interfaccia web.

CVE-2016-8867

Il Docker Engine nella versione 1.12.2 abilitava le capabilities di ambiente con le politiche configurate impropriamente. Questo ha consentito alle immagini dannose di ignorare le autorizzazioni utente per accedere ai file all'interno del filesystem del container o dei volumi montati, causando una notevole divulgazione delle informazioni, ma senza alcun impatto sull'integrità del sistema e sulle risorse dello stesso. La complessità di tale attacco non è elevata, quindi non sono richieste necessarie competenze in ambito della sicurezza e non è richiesto alcun meccanismo di autenticazione per sfruttare tale vulnerabilità.

CVE-2016-9962 (Docker privilege escalation vulnerability)

RunC in Docker, ha permesso che processi mandati in esecuzione all'interno di container attraverso l'utilizzo del comando 'run exec', potessero essere tracciati dal processo PID1 mediante l'utilizzo della syscall ptrace(). Questo ha consentito ai processi principali del container (nel qual caso fossero in esecuzione con i privilegi di root) di accedere ai descrittori dei file di questi nuovi processi durante l'inizializzazione, dando a tali processi la possibilità di uscire all'esterno del container o di modificare lo stato di runC prima che tali processi venissero completamente posizionati all'interno del container. Tale attacco comporta una totale divulgazione delle informazioni presenti nel sistema, con un impatto parziale sull'integrità del sistema. Infatti è possibile modificare

solo alcuni file o informazioni di sistema senza che l'attaccante abbia il controllo su cosa può essere modificato, limitando così ciò che lo stesso può influenzare. Le prestazioni possono essere ridotte a livello di causare interruzioni nella disponibilità delle risorse. Non è necessario alcun meccanismo di autenticazione per sfruttare tale vulnerabilità, ma per portare a segno tale attacco sono necessarie delle competenze più elevate nell'ambito della sicurezza.

CVE-2017-11468 (Docker drop capabilities vulnerability)

Nelle versioni precedenti alla 2.6.2 il repository di Docker non limitava correttamente la quantità di contenuti che poteva essere accettata da un utente, consentendo quindi agli utenti malintenzionati di portare a segno attacchi remoti che causassero un denial of service portando all'esaurimento di tutta la memoria tramite il manifest endpoint. Tale vulnerabilità non ha alcun impatto sulla riservatezza delle informazioni e sull'integrità del sistema. Vi è però una elevata riduzione delle prestazioni con conseguente interruzione della disponibilità delle risorse hardware. Non sono necessarie competenze in ambito della sicurezza per portare a segno tale attacco, e non è necessario alcun meccanismo di autenticazione per sfruttare tale vulnerabilità.

Le vulnerabilità menzionate precedentemente non presentano alcuni exploit secondo *L'Exploit Database* [49] e secondo il CVE - *Common Vulnerabilities and Exposures* [48], il quale per ciascuna vulnerabilità indica che non sono presenti dei moduli *Metasloit*.

LXC, LXD, Docker, Rkt known CVE vulnerabilities.

docker

CVE-ID	Publish date	Score	Complexity	Authentication
CVE-2017-7297	2017-03-28	6.5	Low	Single system
CVE-2016-9962	2017-01-31	4.4	Medium	Not required
CVE-2016-8867	2017-10-28	5.0	Low	Not required
CVE-2016-6595	2017-01-04	4.0	Low	Single system

LXC

CVE-ID	Publish date	Score	Complexity	Authentication
CVE-2017-5985	2017-03-14	2.1	Low	Not required
CVE-2016-10124	2017-01-09	5.0	Low	Not required
CVE-2016-8649	2017-05-01	9.0	Low	Single system

LXD

CVE-ID	Publish date	Score	Complexity	Authentication
CVE-2016-1582	2016-06-09	2.1	Low	Not required
CVE-2016-1581	2017-06-09	2.1	Low	Not required

rkt

CVE-ID	Publish date	Score	Complexity	Authentication

Tabella 2.1. LXC, LXD, Docker, Rkt known CVE vulnerabilities.

Capitolo 3

Attuali soluzioni di sicurezza per i container

In primo luogo, farò un rapido riepilogo dei container rispetto alle macchine virtuali, così da poter introdurre con più semplicità le attuali soluzioni di sicurezza presenti nei container. Ad oggi, i container sono considerati secondo alcuni, la fase successiva dell'evoluzione del software deployment, e tale tecnolocia si presuppone possa arrivare in futuro a soppiantare le macchine virtuali. Secondo altri invece, i container risultano essere un'alternativa che andrebbe scoraggiata rispetto alle macchine virtuali, e se ne consiglia l'utilizzo solo quando e dove le circostanze stesse ne garantiscano il loro utilizzo. Infatti, mentre una macchina virtuale è efficacemente incapsulata e isolata dal sistema host (contenente sia il sistema operativo installato su virtual machine al cui interno viene poi installato ed eseguito un qualunque tipo di software), un containter invece ospiterà il solo software o applicazione da mandare in esecuzione, senza il payload aggiuntivo del sistema operativo della macchina virtuale, basandosi in tal modo sul kernel del sistema operativo presente sulla macchina host e condividendo tale kernel con gli altri container. Con le macchine virtuali invece, si ha anche la presenza di un ulteriore strato di isolamento, il cosiddetto hypervisor che si occupa di gestire la distribuzione, funzionando contemporaneamente come una specie di "firewall" tra la VM e il sistema host sottostante, mentre se si fa riferimento alla seconda tecnologia, il container engine, gestirà non solo la distribuzione, ma anche l'accesso di ciascun container alle risorse host. Il vantaggio più evidente dei contsiner risulta nel fatto che siano molto più leggeri rispetto alle VM, data la mancanza di overhead aggiunto fornito dal sistema operativo presente sulla VM. Quindi i container altre a non richiedere necessariamente la presenza di un sistema operativo guest, conterranno come payload aggiuntivo, solo le dipendenze richieste dall'applicazione che dovrà essere mandata in esecuzione su un dato container. Inoltre i container occupano meno spazio nella memoria rispetto alle macchine virtuali risultando anche più veloci dal punto di vista implementativo. Un container può anche essere generato per uno scopo ben specifico, ed in seguito distrutto immediatamente dopo aver portato a termine l'esecuzione dell'operazione richiesta.

Dal punto di vista della sicurezza, il principale svantaggio dei container rispetto alle macchine virtuali è che essi dipendono sostanzialmente dalle risorse del sistema host sottostante. Un container engine, si occupa quindi di gestire le interazioni tra il container e il sistema operativo host, ma non si presta facilmente al tipo di funzioni fornite dall'hypervisor nell' ambito delle VM. C'è anche la questione della cosiddetta "maturità tecnologica" . Infatti la tecnologia delle macchine virtuali è stata testata abbastanza a lungo, così che la maggior parte dei suoi problemi di sicurezza di base sono stati definiti, analizzati e indirizzati. I container invece, sono ancora abbastanza nuovi, affinché alcune questioni di sicurezza risultano essere ancora in una fase di definizione e analisi.

Per descrivere le varie soluzioni di sicurezza 'nativa e non', presenti nei sistemi di gestione dei container engine, vorrei discutere un'analogia sui meccanismi di sicurezza allo scopo di poter evidenziare l'importanza della sicurezza applicata su più livelli. Infatti prenderò in considerazione un castello che presenta più livelli di difesa utilizzati al fine di poter evitare diversi tipi di attacchi. Tipicamente un castello sfrutta la geografia locale come primo mezzo di difesa, inoltre è costruito

con pareti molto spesse e robuste, ci sono torri di vedetta e difesa e più livelli di mansioni per coloro che stanno in difesa all'interno del castello. Quindi un attaccante dovrebbe superare più di un livello di difesa per poter entrare all'interno di esso e nel caso in cui riuscisse nell'intento, si ritroverebbe a dover affrontare anche le difese all'interno presenti all'interno. In egual modo, le difese di un sistema devono essere implementate su più livelli. I container infatti, dovrebbero essere avviati all'interno di una macchina virtuale, in modo tale che un attacco andato a buon fine in un container non permetta l'accesso al sistema operativo della macchina host. Inoltre, l'utilizzo di un hypervisor sarebbe uno strato di isolamento aggiuntivo tra la VM ed il Sistema operativo della macchina host. Un buon isolamento tra i container si può ottenere utilizzando user namespaces e cgroups per limitare le risorse hardware e questo sarebbe un altro meccanismo di difesa aggiuntivo. L'installazione di sistemi di monitoraggio in caso di comportamenti insoliti e l'utilizzo di firewall per limitare l'accesso alla rete ai container se non strettamente necessario, permetterebbe di ridurrebbe notevolmente la superficie di attacco. Un altro importante principio da rispettare per la sicurezza dei container è quello di abbassare il livello di privilegi alle applicazioni in esecuzione all'interno dei container. Ovvero tali applicazioni dovrebbero essere eseguite con i soli privilegi necessari a svolgere la loro funzione.

Dovrà quindi essere buona regola:

- Assicurarsi che i processi nei container non vengano eseguiti come root.
- I file system andrebbero eseguiti in sola lettura in modo da evitare che un attaccante possa andare a sovrascrivere i dati o salvare degli script dannosi.
- Limitare le chiamate di sistema che un applicazione può effettuare al kernel, al fine di ridurne la superficie di attacco.
- Limitare le risorse che un container può utilizzare, per evitare che un'applicazione se compromessa possa consumare un elevato numero di risorse tale da portare l'intero sistema in blocco a causa di un *Denial Of Service* (DoS). Meccanismi di sicurezza quali CGroups e user namespaces forniscono una protezione nativa sui vari container engine, garantendo l'isolamento tra i vari container e l'host in namespaces differenti e permettendo di assegnare il corretto quantitativo di risorse hardware a ciascuno di essi, per evitare attacchi di denial of services sul sistema host sfruttando vulnerabilità note. Di seguito descriverò i meccanismi di protezione forniti nativamente dalle varie tecnologie di gestione dei container.

Limitazione delle risorse CPU e della memoria

Se un attaccante riesce ad accedere ad un container, per poi andare ad utilizzare risorse CPU, può arrivare a saturare le risorse dell'intero sistema e di tutti gli altri container in avvio sul server, se esse non sono state adeguatamente limitate, generando così un attacco di denial of service sulla macchina. In gestori di container tipo Docker, le risorse CPU vengono assegnate e ripartite equamente per il numero di container avviati sulla macchina host.

La limitazione della memoria invece, protegge dagli attacchi DoS e dalle applicazioni con perdite di memoria che consumano lentamente tutta la memoria dell'host (tali applicazioni possono essere riavviate automaticamente per mantenere un livello di servizio).

Limitazione del riavvio dei container

Se un container continua a fermarsi e riavviarsi, sprecherà una grande quantità di tempo e risorse di sistema, forse nella misura in cui provoca un DoS. Questo ad esempio, utilizzando Docker, può essere facilmente impedito utilizzando la politica di on-failure restart anziché la policy always. In questo modo Docker riavvia il container fino ad un numero finito di volte. Inoltre Docker utilizza un back-off esponenziale quando riavviano i container. Infatti ad esempio, si attenderà 100 ms, poi 200 ms, poi 400 ms e così via sui successivi riavvii. Da solo, questo dovrebbe essere efficace per prevenire gli attacchi DoS che tentano di sfruttare la funzionalità di riavvio.

Limitazione dell'accesso ai file system

Questo fornisce la possibilità di fermare eventualmente, attaccanti che cercano di scrivere in file, pregiudicandogli dunque la possibilità di scrivere uno script e ingannare l'applicazione in esecuzione o di sovrascrivere dei dati sensibili o dei file di configurazione.

Limitazione delle Capabilities

Il kernel Linux definisce dei set di privilegi chiamati capabilities che possono essere assegnati ai processi per fornire loro un maggiore accesso al sistema. Le capabilities coprono un'ampia gamma di funzioni, dalla modifica del tempo di sistema all'apertura di socket di rete. Questo risultava essere particolarmente preoccupante nel caso si volessero utilizzare comandi/utility come il ping, che richiedeva privilegi di root solo per l'apertura di un socket di rete privata. Questo implicava che un piccolo bug nell'utility ping portava a consentire agli attaccanti di ottenere i privilegi di root completi sull'intero sistema. Ad oggi questo problema è stato risolto fornendo all'utility ping solo i privilegi necessari per la creazione di un socket di rete, piuttosto che fornire i privilegi di root globali. Il che significa che un attaccante non avrebbe più la possibilità di sfruttare la precedente vulnerabilità dell'utility ping. Come impostazione predefinita, i gestori di container come ad esempio Docker, vengono eseguiti con un sottoinsieme di funzionalità minime. Infatti un container di default non può utilizzare dispositivi come la GPU e la scheda audio o inserire i moduli del kernel. In termini di sicurezza, l'intento sarebbe quello di limitare le capacità dei container per quanto questo risulti possibile. È inoltre possibile controllare le funzionalità disponibili in un container utilizzando appositi comandi forniti dai containe engine. Ad esempio, se si volesse modificare l'ora di sistema non sarebbe possibile a meno che non si aggiungano i privilegi al container per effettuare la chiamata di sistema di tipo SYS_TIME. Poiché l'ora del sistema è una funzionalità del kernel, l'impostazione del tempo all'interno di un container imposta l'ora anche nell'host e in tutti gli altri container. Un approccio più restrittivo potrebbe essere quello di eliminare tutti i privilegi e aggiungere solo quelli strettamente necessari, portando quindi ad un importante aumento della sicurezza dei container. Infatti un attaccante che riuscisse a portare a buon fine un attacco al kernel di sistema, risulterebbe ancora molto limitato in quanto il kernel farà solo ciò che è in grado di fare o più precisamente svolgerà solo i compiti per il quale è stato settato.

Limitare le risorse di rete tra i container

Limitare le risorse di rete tra i vari container a quelle strettamente necessarie è un altro accorgimento che bisogna applicare per evitare che un attaccante possa spostarsi liberamente tramite i vari container avviati sull'host machine. E' quindi importante bloccare tutte le porte non utilizzate nei vari container. Uno utile strumento per verificare i servizi di rete attivi è netcat.

Applicazione di limiti alle risorse e isolamento dei container in hosts

Il kernel Linux definisce i limiti delle risorse che possono essere applicati ai processi come limitare il numero di processi figlio che possono essere forgiati e il numero di file descrittori aperti consentiti. Questi limiti possono anche essere applicati ai contsiner, sia passando degli appositi flag (tipo -ulimit in Docker) o impostando le impostazioni predefinite per il container.

Inoltre, può essere molto utile nel caso in cui sul server sia in esecuzione un'implementazione multi-tenancy (esecuzione di container per diversi utenti). Ovvero sarebbe necessario utilizzare una macchina virtuale per ciascun utente (tutte insieme gestite da un hypervisor), e al loro interno avviare le applicazioni utente in dei container. Questa sarebbe una soluzione molto meno efficiente in termini di risparmi delle risorse hardware, ma garantirebbe una maggiore sicurezza in ambito multi-tenancy.

Image Provenance and Secure hash

Verificare la provenienza di ogni immagine prima di installarla sul server è un importante passo che permette di utilizzare le immagini in modo sicuro. E' importante che si stia installando la stessa immagine che ha verificato lo sviluppatore, in modo da evitare di avviare immagini sui server, contenenti del codice malevolo che potrebbe garantire un pieno accesso alla macchina host. Uno strumento che permette di stabilire la provenienza di un software, è basato sull'utilizzo di una chiave di hash, che presenta caratteristiche molto simili ad un'impronta digitale, ma è applicata ai dati. L'hash è una stringa relativamente piccola che è unica per i dati forniti. Ogni modifica dei dati provocherà una modifica all'hash stesso. Nel caso in cui si sia in possesso di un hash sicuro per dei relativi dati, è possibile ricalcolare tale hash sulla base dei dati scaricati e confrontarlo con l'originale. Nel caso in cui non vi sia alcuna corrispondenza tra gli hash, questo implicherà che i dati sono stati danneggiati durante il download o upload sul registry o ancor peggio, sono stati manomessi al fine di inserire del codice malevolo. Sono disponibili diversi algoritmi per il calcolo dell'hash e ognuno con un diverso grado di complessità e garanzie di unicità dell'hash stesso. Gli algoritmi principlmente utilizzati sono SHA (che presenta diverse varianti dello stesso algoritmo) ed MD5 (che risulta obsoleto e quindi presenta diversi problemi per il quale andrebbe evitato e sostituito con il precedente). Bisogna inoltre aggiungere che se si effettua la verifica di un hash e vi è un'esatta corrispondenza, si può stare tranquilli che i dati non siano stati danneggiati o manomessi, ma rimane un fondamentale problema, ovvero che l'hash con il quale si sta facendo il confronto non è detto che sia originale e che non sia stato sostituito con un altro da un utente malevolo, oltre ovviamente ad aver modificato i dati stessi che sono protetti da hash. Una soluzione a questo problema è data dalla possibilità di applicare una firma crittografica con coppie di chiavi pubbliche/private. Attraverso tale metodo infatti è possibile verificare l'identità dello sviluppatore e della propria applicazione. Quindi se un editore firma una qualunque applicazione utilizzando una sua chiave privata, qualsiasi utente che vorrà utilizzare l'applicazione in questione, potrà verificare che essa provenga effettivamente da un dato sviluppatore, utilizzando la chiave pubblica per controllare la firma dello sviluppatore. Quindi supponendo che un utente abbia già ottenuto una copia della chiave e che essa non sia stata manomessa, in caso di corrispondenza si potrà quindi essere sicuri che il contenuto che si andrà ad utilizzare proviene realmente da quello sviluppatore.

Auditing

L'esecuzione di regolari revisioni (Auditing) sui propri container e sulle immagini è un ottimo modo per assicurare che il sistema venga mantenuto pulito, aggiornato e per verificare che non siano state riscontrate violazioni nell'ambito della sicurezza. Un buon controllo in un sistema basato su container dovrebbe verificare che tutti i container in esecuzione stiano utilizzando immagini aggiornate e che queste immagini utilizzino a loro volta un software aggiornato e sicuro. Ogni divergenza o anomalia in un container rispetto all'immagine dal quale il container stesso è stato creato, dovrebbe essere identificata e controllata. Inoltre, gli audit dovrebbero riguardare anche altre aree non specifiche di sistemi basati su container, come ad esempio, sarebbe importante controllare i registri di accesso, le autorizzazioni di file e l'integrità dei dati. Una buona automatizzazione degli audit è molto utile per rilevare eventuali problemi ed allo stesso tempo nel modo più rapido possibile, invece di dover effettuare l'accesso a ciascun container al fine di esaminare ciascuno di essi in modo individuale. Inoltre è possibile verificare l'immagine utilizzata per costruire il container. Questo meccanismo funziona ancora meglio se si utilizza un file system in sola lettura per essere sicuri che nulla sia cambiato all'interno del container. La quantità di lavoro che riguarda il controllo può essere seriamente ridotta eseguendo immagini minime che contengono solo i file e le librerie essenziali per l'applicazione stessa. Il sistema host dovrà inoltre essere sottoposto a revisione come una normale macchina host o VM. Inoltre è necessario assicurarsi che il kernel sia correttamente patchato, in particolar modo in un sistema basato su container dove il kernel è condiviso tra di essi.

3.1 Add-on e soluzioni di sicurezza

Ad oggi sono presenti diversi strumenti 'nativi e non ', per il controllo e la gestione della sicurezza dei sistemi basati su container e che di seguito descriverò.

3.1.1 Namespaces

Il principio dei namespaces [1] si basa sulla definizione di un'etichetta (namespace) che viene associata ad un gruppo di risorse di sistema come ad esempio i mountpoints, al quale assegniamo uno o più processi decidendo quali risorse questi processi possono vedere. L'utilizzo dei namespaces in Linux si basa sull'uso di tre system calls specifiche:

- syscall clone() crea processi o thread;
- syscall unshare() separa le risorse di un processo figlio dal padre;
- syscall setns() specifica dei namespaces.

Come sempre, tutte le syscall si possono anche chiamare con appositi comandi da shell. Quando si clona un processo mediante l'utilizzo di clone() è possibile specificare, impostando alcuni flag, se il namespace delle varie categorie (user, net, PID, etc) debba essere o meno condiviso con il processo padre. Per chiarire, se dichiaro che il processo figlio non deve condividere il namespace dei mountpoint, il kernel duplicherà la struttura dei filesystem visibili al processo figlio (clonato) e da quel momento in poi qualsiasi operazione di mount() e umount() effettuata dal processo figlio non avrà effetto sui filesystem visti dal processo padre. In teoria, impostando tutti i flag nella maniera appropriata risulterà possibile isolare il processo figlio in un ambiente separato in cui tutte le risorse del sistema operativo saranno visibili solo ad esso.

La chiamata di sistema unshare() permette di "scollegare" un set di risorse dal processo padre. Sarà quindi possibile ottenere lo stesso risultato ottenuto con l'utilizzo di clone(), impostando un namespace differente, ma a runtime e senza dover chiamare un'altra volta clone(). Questo perché non tutti i flag possono essere impostati a runtime.

L'ultima chiamata setns() è stata pensata per l'utilizzo dei namespaces e consente di assegnare un processo ad uno specifico namespace (in una delle categorie sottoelencate), utilizzando il descriptor del namespace (si trovano sotto /proc/[PID]/ns).

Ci sono sei categorie che compongono le risorse di sistema dei namespaces:

- PID
- IPC
- UTS
- Mount points
- Network
- Users

È quindi abbastanza facile intuire che le chiamate appena illustrate, ci permettono di condividere le risorse di sistema tra i vari processi in esecuzione. Questo è il concetto alla base del funzionamento della virtualizzazione a container.

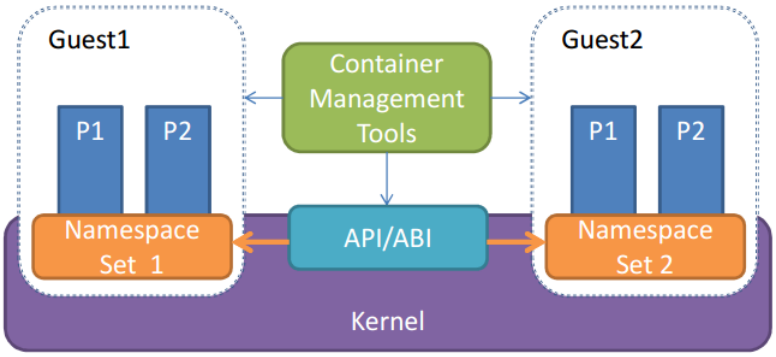

Figura 3.1. Kernel Namespaces [25].

3.1.2 CGroups

Un control group è un insieme di processi che sono legati dagli stessi criteri. Questi gruppi possono essere gerarchici, in cui ogni gruppo eredita dei limiti dal proprio gruppo di appartenenza. Il kernel quindi, si dovrà occupare di fornire l'accesso a più controllori (sottosistemi) attraverso l'interfaccia di cgroups.

L'infrastruttura di cgroups fornisce solo le funzionalità di raggruppamento dei processi, mentre i vari sottosistemi di cgroups attuano le politiche di controllo specifico per ogni tipo di risorsa. Questo framework è molto potente e permette di impostare le regole di risorse non solo sulla base di utenti e gruppi.

Per esempio si potrebbe voler prevenire che un server web utilizzi tutta la memoria su un sistema che ha anche in esecuzione un database.

Quindi si andranno ad allocare le risorse di sistema tra gruppi di utenti con priorità differenti (ad esempio, gli ospiti il 10

CGroup può essere utilizzato inoltre anche per isolare e dare comandi speciali a gruppi di processi. Per cui possiamo dire che ci sono 2 tipi di sottosistemi;

I sottosistemi per l'isolamento e controlli speciali I sottosistemi per il controllo delle risorse

Inizialmente ci si focalizzerà sul controllo delle risorse fornendo quindi una piccola panoramica dei sottosistemi per l'isolamento e controlli speciali:

CPUset: assegna singole CPU e nodi di memoria a cgroups. Namespace : fornisce una visione privata del sistema per i processi in un cgroup, ed è usato principalmente per realizzare la virtualizzazione a livello di sistema operativo. Freezer: ferma tutti i processi in esecuzione in un cgroup, rimuovendoli dal task scheduler del kernel . Device: consente o nega l'accesso ai dispositivi ai processi in un cgroup. Checkpoint/Restart: Salva lo stato di tutti i processi in un cgroup in un file di dump. E' possibile riavviare i processi in un secondo momento, o solamente salvarne il loro stato e continuare.

L'amministrazione dei cgroup viene quindi effettuata per mezzo di uno special virtual filesystem che consiste in una sorta di procfs o sysfs, e che e banalmente denominato cgroupfs.

3.1.3 Signature verification

Come citato nell'articolo [56] una richiesta comune da parte delle comunità che si occupano dello sviluppo di container engine, in particolar modo la comunità di Docker, è la necessità di avere garanzie crittografiche forti su quale codice e quali versioni di software vengono eseguite nell'infrastruttura. Sono quindi necessari dei meccanismi per verificare l'editore di un'immagine. Prima che un editore effittui l'upload delle proprie immagini nei registri pubblici o cosiddetti repository, è necessario che tale immagine venga firmata localmente con l'ausilio di una chiave

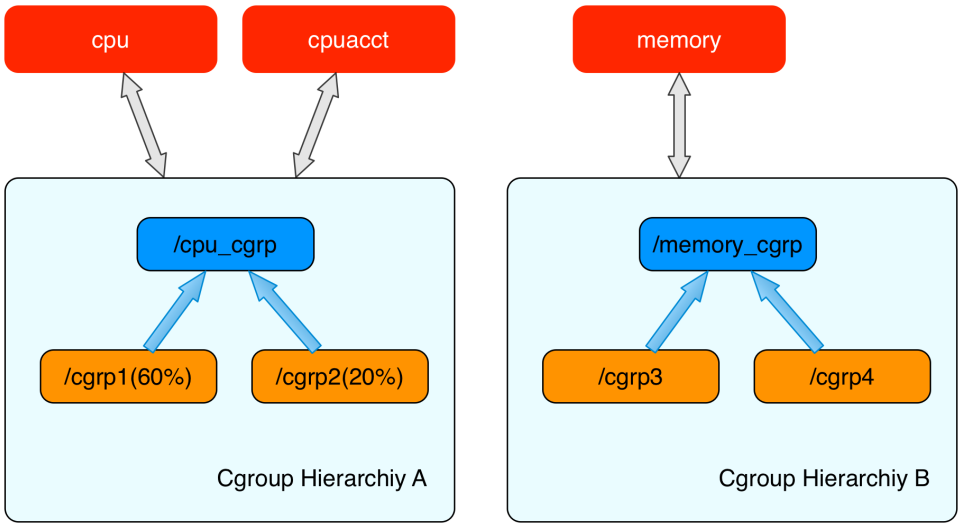

Figura 3.2. Control Groups.

privata appartenente all'editore della stessa. In seguito, quando un consumer deciderà di scaricare questa immagine, il container engine utilizzerà la chiave pubblica del publisher, per verificare che l'immagine che andrà eseguita, sia effettivamente quella che l'editore ha creato e che quindi non sia stata manomessa in alcun modo. Quindi tutte le operazioni che andranno ad utilizzare un registro remoto, faranno uso di immagini firmate e verificate, al fine di impedire possibili manomissioni dei contenuti delle immagini presenti nei repository, da parte di utenti malevoli.

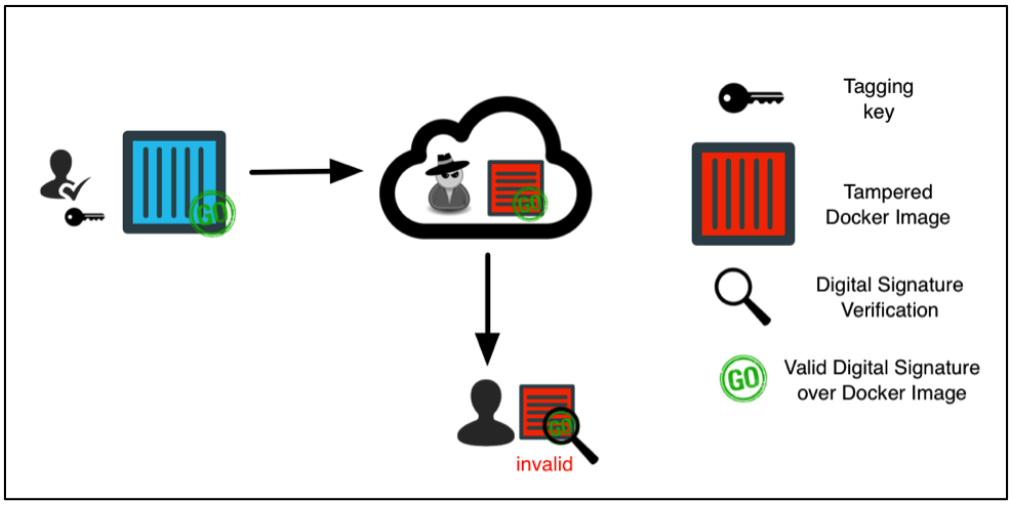

Figura 3.3. Signature Verification [26].

Content Trust

Con il trust del contenuto abilitato, tutte le operazioni che utilizzano un registro remoto impongono l'utilizzo di immagini firmate e verificate. Quindi i comandi CLI di docker che operano sulle immagini con tag, devono avere il contenuto firmato o l'hash del contenuto esplicito. I comandi che operano con il content trust sono:

- push

- build
- create
- pull
- run

Con il content trust abilitato, docker può ad esempio effettuare il pull di un'immagine dal repository ufficiale. L'esito sarà positivo solo se l'immagine avrà il contenuto firmato. Quindi con il Content Trust abilitato, è pssobile evitare attacchi di tipo replay, i quali risultano essere un problema comune nella progettazione di sistemi distribuiti sicuri, in cui i payload precedentemente validi vengono replicati allo scopo di poter ingannare un altro sistema. Lo stesso problema esiste nei sistemi di aggiornamento software, dove le vecchie versioni del software firmato possono essere presentate come le più recenti. Infatti, se un utente viene ingannato nell'installare una versione precedente di un determinato software, l'utente malintenzionato potrà utilizzare le vulnerabilità di sicurezza note per poter in seguito compromettere l'host della vittima. Il Content Trust infatti, utilizza il Timestamp key durante la pubblicazione dell'immagine, fornendo protezione contro gli attacchi di tipo replay e garantendo quindi che l'utente riceva il contenuto più aggiornato. Un altro vantaggio è di garantire protezione contro il Key Compromising.

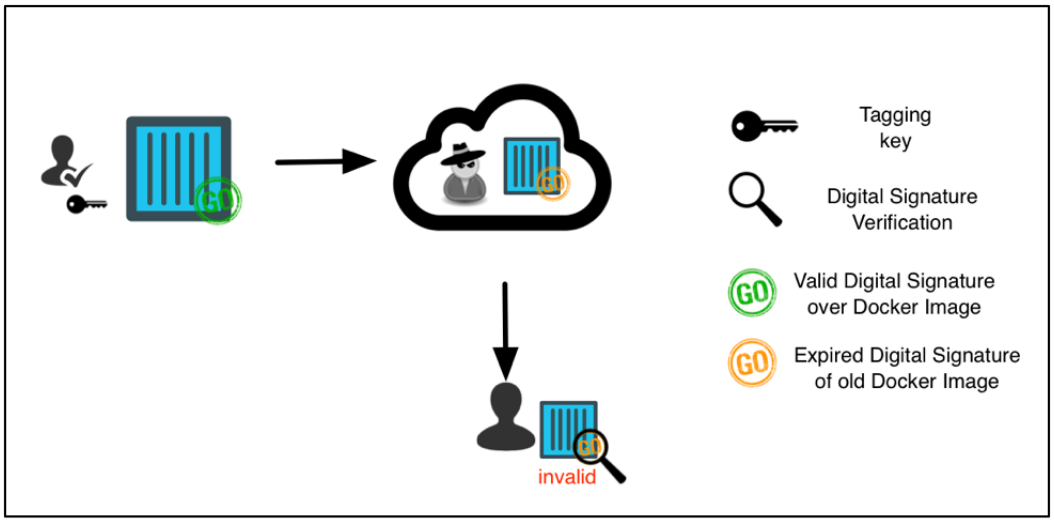

Figura 3.4. Content Trust Timestamp Key [26].

Come mostrato nello scenario presente in figura 3.5 , la chiave di Tagging Online, potrebbe essere compromessa. La chiave di Tagging Online quindi, risulta necessaria ogni volta che viene effettuato il pushing di un nuovo contenuto relativo all'immagine che ha associata la chiave di Tagging in questione su uno specifico repository. Questo però, rende tale chiave intrinsecamente più esposta rispetto alla chiave di Tagging Offline. Il Content Trust quindi, utilizza *The Update Framework* (TUF) per separare le responsabilità attraverso una gerarchia di chiavi in modo che la perdita di una particolare chiave di per sé non sia fatale per la sicurezza del sistema.

The Update Framework (TUF)

L'Update Framework è quindi un design generale sicuro per il problema della distribuzione e degli aggiornamenti del software. TUF infatti, risolve il problema di aiutare gli sviluppatori a proteggere i sistemi di aggiornamento software fornendo una struttura di sicurezza flessibile che consente alle applicazioni di essere sicure da tutti gli attacchi noti al processo di aggiornamento del software. L'utilizzo di TUF offre vantaggi di Docker Content Trust attorno alla compromissione delle chiavi superstiti, la possibilità di avere aggiornamenti affidabili su mirror non protetti, protezione da attacchi di replay, e attacchi di downgrade.

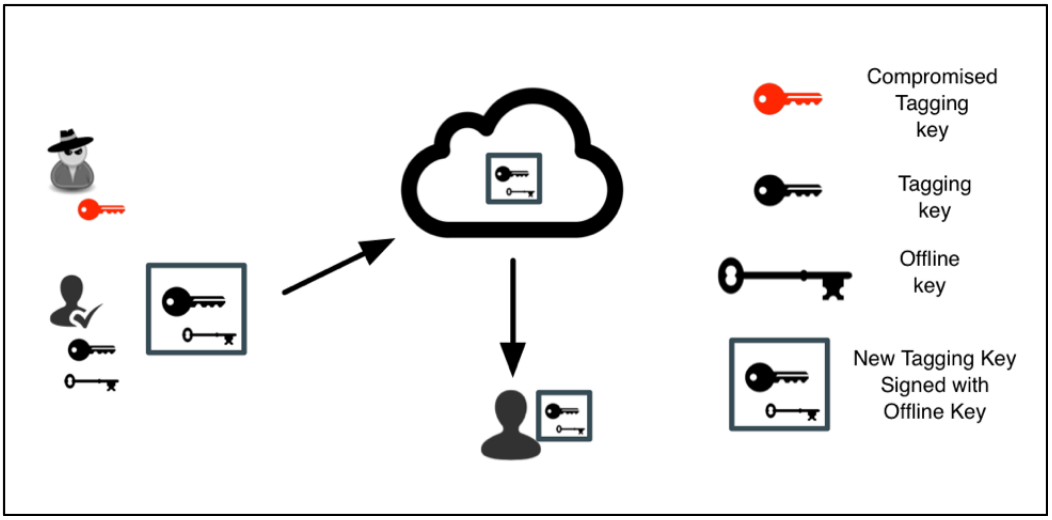

Figura 3.5. Key Comporomising Protection [26].

3.1.4 Discrectionary Access Control

Discretionary access control (DAC) è un tipo di controllo di accesso di sicurezza che concede o limita l'accesso agli oggetti tramite un cosiddetto criterio di accesso determinato dal gruppo proprietario di un oggetto e/o dai soggetti stessi. I controlli del meccanismo DAC sono definiti dall'identificazione dell'utente con le credenziali fornite durante l'autenticazione, come nome utente e password. Il DAC è discrezionale in quanto il soggetto (proprietario), potrà trasferire oggetti autenticati o l'accesso alle informazioni ad altri utenti. In altre parole, il proprietario avrà il compito di determinare i privilegi di accesso agli oggetti stessi. Nel DAC quindi, ogni oggetto di sistema (file o oggetto dati) ha un proprietario e ciascun proprietario dell'oggetto iniziale sarà l'oggetto che ne causerà la creazione. E' per tale motivo che il criterio di accesso di un oggetto viene determinato dal suo proprietario. Tale proprietario, avrà in questa implementazione DAC anche la possibilità di poter decidere e/o assegnare attributi di sicurezza.

Un tipico esempio del DAC è la modalità file Unix, che definisce le autorizzazioni di lettura, scrittura ed esecuzione in ciascuno dei tre bit, questo per ciascun utente, gruppo e altri. Quindi, quando ad esempio si vuole creare un file, si decide quali privilegi di accesso si desidera dare ad altri utenti nel caso in cui essi vogliano accedere a questo determinato file. A questo punto, il sistema operativo effettuerà la decisione sul controllo degli accessi in base ai privilegi di accesso che sono stati precedentemente impostati.

Quindi il DAC con i suoi attributi, permetterà che:

- L'utente possa trasferire la proprietà dell'oggetto ad un altro utente.
- L'utente potrà determinare il tipo di accesso di altri utenti.
- Dopo diversi tentativi, i fallimenti delle autorizzazioni limiteranno l'accesso degli utenti.
- Gli utenti non autorizzati saranno limitati dal vedere le caratteristiche dell'oggetto, come ad esempio la dimensione del file, il nome del file e il percorso della directory che contiene tale file.
- L'accesso agli oggetti verrà determinato durante l'autorizzazione dell'elenco di controllo di accesso o *Access Control List* (ACL) e in base all'identificazione dell'utente e/o all'appartenenza al gruppo.

3.1.5 Seccomp

Seccomp acronimo di Secure Computing Mode è una funzionalità del kernel che consente di filtrare le chiamate di sistema provenienti dal container verso il kernel. La combinazione di chiamate limitate e chiamate consentite è disposta in profili, ed è possibilie passare questi diversi profili a diversi container. Seccomp fornisce inoltre un controllo delle funzionalità ad una granularità più alta, dando ad un attaccante un numero limitato di syscall che possono essere chiamate dall' interno di un container. Il profilo di default di seccomp è costituito da un file JSON che permette di bloccare 44 chiamate di sistema in riferimento a più di 300 disponibili. Vi è anche un profilo più rigoroso che risulta essere un miglior compromesso con la compatibilità delle applicazioni, e tale profillo blocca 57 chiamate di 300. Seccomp inoltre utilizza il sistema Berkeley Packet Filter (BPF), programmabile in modo da poter ottenere un filtro personalizzato. È anche possibile limitare una determinata syscall anche personalizzando le condizioni per come o quando questa syscall dovrebbe essere limitata. Un filtro seccomp si occupa quindi di sostituire una specifica syscall con un puntatore a un programma BPF che eseguirà quel dato programma anziché la syscall. Tutti i figli di un processo con questo filtro erediteranno anche tale filtro. Attualmente, Linux supporta i seguenti valori operativi.

`SECCOMP_SET_MODE_STRICT`

Il sistema chiede che al thread chiamante sia consentito di effettuare syscall di tipo read, write, _exit (ma non exit_group) e sigreturn. Altre chiamate di sistema comportano l'inoltro di un segnale di tipo SIGKILL. La modalità STRICT è utile per applicazioni che hanno necessità di eseguire codice non fidato ottenuto tramite un socket o una pipe.Inoltre, anche se il thread di chiamata non è più in grado di chiamare sigprocmask, può sempre utilizzare sigreturn per bloccare tutti i segnali eccetto SIGKILL e SIGSTOP. Ciò significa che per terminare in modo affidabile il processo, deve comunque essere utilizzato SIGKILL. Questo può essere fatto utilizzando timer_create con SIGEV_SIGNAL impostato su SIGKILL, o utilizzando setrlimit per impostare il limite più forte per Rlimit_cpu. Questa operazione è disponibile solo se il kernel è configurato con CONFIG_SECCOMP abilitato. Il valore delle flag deve essere 0 e args deve essere NULL. Inoltre questa operazione risulta essere funzionalmente identica alla chiamata: prctl (PR_SET _SECCOMP, SECCOMP_MODE_STRICT).

`SECCOMP_SET_MODE_FILTER`

Nella modalità FILTER le chiamate di sistema consentite sono definite da un puntatore a Berkeley Packet Filter (BPF) che viene poi passato tramite args. Questo argomento è un puntatore di una struct sock_fprog, che può essere progettato per filtrare le chiamate di sistema arbitrarie e gli argomenti di chiamata di sistema. Se il filtro non è valido, seccomp() fallisce,e restituisce EINVAL inel campo errno. Inoltre se fork o clone sono consentiti dal filtro, ogni processo figlio bambino sarà vincolato agli stessi filtri di chiamata di sistema del genitore. Invece se viene consentito execve, i filtri esistenti saranno conservati attraverso una chiamata execve. Per poter utilizzare l'operazione SECCOMP_SET_MODE_FILTER, il chiamante deve disporre della CAP_SYS_ADMIN nel proprio userspace oppure il thread deve già avere il no_new_privs bit settato ad 1. Se quel bit non era già stato impostato da un antenato di questo thread, il thread deve effettuare la chiamata prctl (PR_SET_NO_NEW_PRIVS, 1). Altrimenti, l'operazione SECCOMP_SET_MODE_FILTER fallirà restituendo EACCES in errno. Questo requisito assicura che un processo senza alcun privilegio di root non può applicare un filtro dannoso e quindi invocare un set-user-ID o un altro programma privilegiato utilizzando execve, che quindi potrebbe potenzialmente compromettere il programma. Come ad esempio anche un filtro dannoso potrebbe causare un tentativo di utilizzo di setuid per impostare gli ID utente del chiamante a valori diversi da zero invece di restituire 0 senza aver invece effettivamente fatto una system call. Quindi, il programma potrebbe essere in grado di effettuare cose pericolose al resto del sistema, poichè manterrebbe i privilegi di superuser attivi invece di disattivarli. Quando il filtro collegato è consentito da prctl o seccomp() possono essere aggiunti altri filtri. Questo porterà ad un incremento del tempo di esecuzione, ma ad una ulteriore riduzione della superficie di attacco, durante l'esecuzione di un thread. L'operazione SECCOMP_SET_MODE_FILTER è disponibile solo se il kernel è configurato con CONFIG_SECCOMP_FILTER abilitato. Quando i flags sono settati a 0, questa operazione risulta funzionalmente identica alla chiamata prctl (PR_SET_SECCOMP, SECCOMP_MODE_FILTER, args).

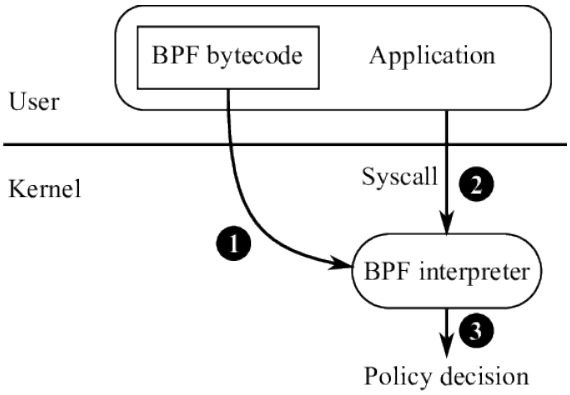

Figura 3.6. Seccomp Architecture. Gli sviluppatori di applicazioni specificano la loro politica di chiamata di sistema come filtro BPF, in forma di bytecode. All'avvio, l'applicazione spazio-utente invia il filtro al kernel. Il kernel invoca un interprete BPF per valutare il programma contro ogni chiamata di sistema successiva e decide se rifiutarlo o rifiutato in base al risultato dell'interprete [27].

3.1.6 Mandatory Access Control

Nel *Mandatory Access Control* (MAC) il sistema (e non gli utenti) specifica quali soggetti possono accedere a specifici oggetti di dati. Il modello MAC infatti si basa su etichette di sicurezza. Ai soggetti, verrà quindi fornito un nulla osta di sicurezza (secret, top secret, confidential, ecc.). Mentre agli oggetti verrà assegnata una classificazione di sicurezza (secret, top secret, confidential, ecc..). I dati di classificazione vengono quindi memorizzati nelle etichette di sicurezza, le quali risultano legate a specifici soggetti e oggetti.

Quando il sistema si troverà nella situazione di dover prendere uuna decisione sul controllo degli accessi, cercherà di far corrispondere l'autorizzazione del soggetto con la classificazione di un determinato oggetto. Ad esempio, se un utente risulta essere in possesso di un'autorizzazione di sicurezza di tipo secret e richiede un oggetto con una classificazione di sicurezza di tipo top secret, ad esso gli verrà negato l'accesso poichè la sua autorizzazione risulta essere di livello inferiore alla classificazione di tipo top secret dell'oggetto. Il modello MAC viene solitamente utilizzato in ambienti in cui la riservatezza risulta essere della massima importanza, come in ambito militare.

Alcuni esempi di sistemi commerciali basati su MAC sono SELinux e AppArmor.

Linux Security Modules

Linux Security Modules (LSM) è basato su un framework che fornisce un meccanismo per i vari controlli di sicurezza che deve essere agganciato dalle nuove estensioni del kernel. Il nome "module " potrebbe risultare inappropriato in quanto queste estensioni non sono in realtà moduli kernel caricabili. Al contrario, sono selezionabili in fase di compilazione tramite CONFIG_DEFAULT_SECURITY e possono essere sovrascritti all'avvio tramite l'argomento della riga di comando del kernel "security = ..." , nel caso in cui più di questi LSM siano stati incorporati in un determinato kernel.

Gli utenti principali dell'interfaccia LSM sono quindi le estensioni MAC che forniscono una politica di sicurezza completa. Come detto nella sezione 3.1.6 alcuni esempi sono SELinux e AppArmor. Oltre alle estensioni MAC, è possibile inoltre, creare altre estensioni utilizzando l'LSM per fornire modifiche specifiche al funzionamento del sistema quando tali modifiche non sono disponibili nelle funzionalità di base di Linux.

Senza uno specifico LSM integrato nel kernel, verrà applicato in Linux quello di default. La maggior parte degli LSM sceglie di estendere il sistema delle capabilities, costruendo i loro controlli in cima.

Un elenco dei moduli di sicurezza attivi può essere trovato leggendo /sys/ kernel/security/lsm. Questo è un elenco che includerà sempre il capability module. L'elenco rifletterà quindi l'ordine

in cui verranno effettuati i controlli. Tale capability module, risulterà quindi essere sempre il primo, per poi essere seguito da eventuali moduli "minori" , come ad esempio "Yama" [47] e poi un modulo "principale" , come ad esempio SELinux nel caso in cui fosse configurato.

3.1.7 AppArmor

Il vantaggio ed a sua volta lo svantaggio di AppArmor è che è molto più semplice di SELinux, poiché non è in grado di offrire una protezione ad un livello di granularità al pari con SELinux. AppArmor ha la funzionalità di applicare dei cosiddetti profili ai processi, limitandone così i privilegi che hanno a livello di funzionalità Linux e limitandone inoltre l'accesso ai file. In un Host Ubuntu soloitamente AppArmor è abilitato di default. Eseguendo il comando sudo apparmor_status, è possibile verificare che esso sia realmente in esecuzione sulla macchina linux. Inoltre, il gestore di container applica automaticamente un profilo di AppArmor a ciascun container in esecuzione. Il profilo applicato di default, fornisce un livello di protezione contro i rogue container che sono stati infettati da malware e che tentano di accedere a varie risorse di sistema. Al momento della scrittura, il profilo predefinito non può essere modificato, in quanto il daemone del container engine lo sovrascriverà al riavvio. Nel caso in cui AppArmor interferisca con l'esecuzione di un container,esso può essere disattivato per quel singolo container. È inoltre possibile passare un profilo diverso per uno specifico cotantainer.

3.1.8 SELinux

Il modulo SELinux o Security Enhanced Linux è stato sviluppato dall'Agenzia Nazionale per la Sicurezza Nazionale (NSA) come implementazione di ciò che chiamano il controllo di accesso obbligatorio (MAC), in contrasto con il modello Unix di controllo di accesso discrezionale (DAC). In un linguaggio piuttosto chiaro, esistono due grandi differenze tra il controllo di accesso eseguito da SELinux e i controlli standard di accesso di Linux:

I controlli SELinux vengono eseguiti in base a dei tipi, che sono essenzialmente etichette applicate a processi e oggetti (file, socket e così via). Se la politica di SELinux vieta un processo di tipo A per accedere a un oggetto di tipo B, tale accesso sarà disattivato, indipendentemente dalle autorizzazioni di file sull'oggetto o dai privilegi di accesso dell'utente. I test SELinux si verificano dopo i controlli di autorizzazione dei file normali.

È possibile inoltre, applicare più livelli di sicurezza, come ad esempio: secret e top-secret. I processi che appartengono a un livello inferiore non possono leggere i file scritti da processi di livello superiore, a prescindere dal dove nel file system il file risiede o quali sono le autorizzazioni del file. Quindi un processo top-secret potrebbe scrivere un file a /tmp con privilegi di chmod 777, ma un processo riservato non sarebbe ancora in grado di accedere al file. Questo è noto come *Multi Level Security* (MLS) in SELinux, che ha anche il concetto di *Multi Category Security* (MCS). MCS consente alle categorie di essere applicate a processi e oggetti e nega l'accesso ad una risorsa se non appartiene alla categoria corretta. A differenza di MLS, le categorie non si sovrappongono e non sono gerarchiche. MCS può essere utilizzato per limitare l'accesso alle risorse a sottotitoli di un tipo (ad esempio, utilizzando una categoria univoca, una risorsa può essere limitata all'utilizzo da un solo processo). SELinux viene installato per impostazione predefinita sulle distribuzioni di Red Hat e dovrebbe essere semplice da installare sulla maggior parte delle altre distribuzioni. È possibile controllare se SELinux sia in esecuzione, eseguendo comando *sestatus*, il quale restituirà in risposta se SELinux è abilitato o disabilitato e se è in modalità Permissive o Enforcing. Quando è in modalità modalità Permissive, SELinux registra le infrazioni al controllo di accesso ma senza imporle. Il criterio predefinito di SELinux per Docker è progettato per proteggere l'host da container, nonché i container da altri container. I container sono assegnati al tipo di processo predefinito svirt_lxc_net_t e i file accessibili a un container vengono assegnati a svirt_sandbox_file_t. La politica impone che i container siano in grado di leggere ed eseguire file solo da / usr sull'host e non possono scrivere su qualsiasi file nell'host. Inoltre assegna a ciascun container un numero di categoria MCS unico, destinato a impedire ai container di accedere a file o risorse scritte da altri container in caso di rottura. Quindi come anticipato in questa sezione, SELinux fornisce tre tipi di enforcement, ovvero il Type Enforcement, il MCS ed infine il MLS, i quali verranno descritti di seguito in modo più approfondito.

Type enforcement

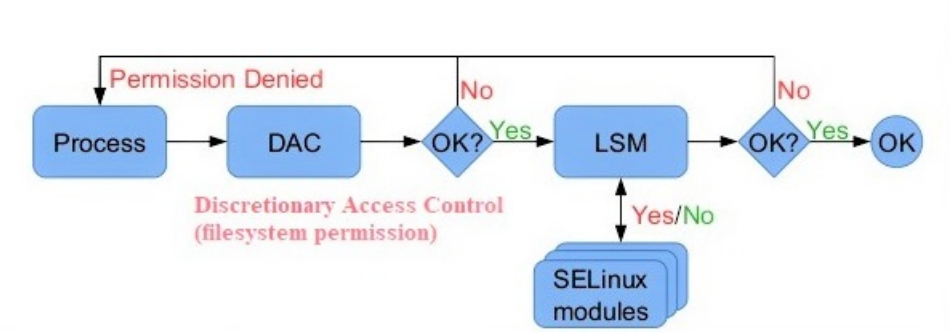

Figura 3.7. LSM [28].

Questo è il modello primario di SELinux che viene più comunemente chiamato type enforcement. Ciò significa che fondamentalmente andremo a definire un'etichetta su un processo basata sul suo tipo e un' ulteriore etichetta su un file system object basata su questo tipo.

Di seguito descriverò un esempio, dove ho mandato in esecuzione due container, il quale uno conteneva l'immagine di un server Apache, e l'altro un database mySQL, ed etichettando il processo Apache in esecuzione sul container1 come httpd_t, e contrassegnando il contenuto Apache come httpd_sys_content_t e httpd_sys_content_rw_t. In seguito ho supposto di avere memorizzati all'interno del database mySQL in esecuzione sull secondo container, dei dati relativi a delle carte di credito è che ho etichettato con msyqld_data_t.

A questo punto, se un processo Apache subisse un attacco malevolo, l'utente malintenzionato potrebbe ottenere il controllo del processo httpd_t e sarebbe autorizzato a leggere i file httpd_sys_content_t e a scrivere su httpd_sys_content_rw_t. Ma l'utente malintenzionato, non avrebbe i permessi per poter leggere i dati della carta di credito che sono stati memorizzati in un database mySQL etichettato come mysqld_data_t. Questo anche se il processo all' interno del container è in esecuzione con i permessi di root. In questo caso SELinux ha attenuato la rottura.

MCS enforcement

Solitamente, nei sistemi informatici sono presenti molti processi in esecuzione tutti con lo stesso tipo di accesso, anche se però si avrebbe la necessità di tenere tali processi separati gli uni dagli altri, come in un ambiente cloud multi-tenancy. La migior soluzione è quella di utilizzare delle macchine virtuali. Infatti, utilizzando un server che ha in esecuzione un certo numero di macchine virtuali e una di esse venisse attaccata, l'utente malintenzionato che ha portato a segno tale attacco, deve essere limitato alla sola VM in questione. Con un sistema type enforcement, etichetto la macchina virtuale KVM con svirt_t e l'immagine come svirt_image_t. In seguito imposto alcune regole che dicono che svirt_t può leggere/scrivere/eliminare i contenuti contrassegnati con svirt_image_t. Con libvirt, ho implementato non solo la separazione delle forme di tipo, ma anche la separazione MCS (Multy Category Security). Infatti quando libvirt sta per lanciare una macchina virtuale, esso sceglie un'etichetta MCS casuale come ad esempio s0: d1, d2, e quindi assegna l'etichetta svirt_image_t: s0: d1, d2 a tutto il contenuto che la macchina virtuale avrà bisogno di gestire. Infine, verrà lanciata la macchina virtuale come svirt_t: s0: d1, d2. Quindi, SELinux controlleranno che svirt_t: s0: d1, d2 non possa scrivere su svirt_image_t: s0: d3, d4, anche se la macchina virtuale venisse controllata da un utente malintenzionato, e anche se questa è in esecuzione don i permessi di root.

Un altro esempio può essere quello di utilizzare una separazione simile in OpenShift. Ogni processo utente/applicazione viene eseguito con lo stesso tipo di etichetta SELinux: openshift_t. La politica che invece definisce le regole che controllano l'accesso del tipo processo utente e un'etichetta MCS unica, utile per assicurarsi che un processo utente non possa interagire con altre applicazioni.

MLS enforcement

Un'altra forma di applicazione SELinux che risulta però poco utilizzata, prende il nome di Multi Level Security (MLS). L'idea principale è quella di controllare i processi basati sul livello dei dati che utilizzeranno. Un processo identificato come "secret" non può leggere dati di tipo "top secret" . MLS è molto simile a MCS, tranne per il fatto che aggiunge un concetto di dominanza all'applicazione. Ovvero, dove le etichette MCS devono corrispondere esattamente, un'etichetta MLS può dominare un'altra etichetta MLS e quindi ottenerne l'accesso.

In un altro esempio ho considerato due server Apache eseguiti in due container differenti container3 e container4, uno dei quali viene eseguito come httpd_t: TopSecret e quello in esecuzione sull'altro container come httpd_t: Secret. Ora suppongo che il processo Apache httpd_t: Secret in esecuzione sul container3 subisca un attacco malevolo. L'utente malintenzionato, riuscirebbe a leggere httpd_sys_content_t: Secret, ma gli verrebbe impedito l'accesso in lettura di httpd_sys_content_t: TopSecret. Invece, se il server Apache in esecuzione che è stato identificato come httpd_t: TopSecret venisse attaccato, fornirebbe all'utente malevolo la possibilità di accedere in lettura ai dati di httpd_sys_content_t: Secret e a quelli di httpd_sys_content_t: TopSecret.

Un uso tipico del Multy Level Security enforcement è in ambito militare, dove un utente può essere abilitato ad accedere a dei dati etichettati come secret, mentre un altro utente di livello superiore, può accedere nello stesso sistema a delle informazioni di livello Top secret ed ovviamente anche a quelle secret.

3.1.9 SELinux e Networking

L'utilizzo di SELinux può essere esteso anche a livello di rete. Infatti ci sono diversi casi in cui più applicazioni comunicano tra di loro attraverso reti locali o remote. Risulta quindi necessario implementare uno o più meccanismi utili a garantire la sicurezza tra le varie componenti di rete. Prendiamo come esempio delle applicazioni che vanno in esecuzione ciascuna all'interno di uno specifico container e tutti i container a loro volta dovranno essere eseguiti all'interno di una rete dedicata. Per tale scopo si vuole che solo che alcuni di questi container possano comunicare direttamente tra di loro e che ad altri venga negato l'accesso diretto a degli specifici container. Se prendiamo come esempio uno specifico ambiente costituido da un Reverse proxy che funzionerà da load balancer sul traffico diretto a due application services, i quali accederanno ad un database comune. In questo caso risulta necessario che il container il quale manda in esecuzione l'immagine del Reverse proxy non possa comunicare in modo diretto con il container che sta eseguendo l'immagine del database e viceversa, ma che solo i due application services possano accedere direttamente a tale servizio. Per fare ciò risulta necessario gestire gli indirizzi IP associati a ciascuno di questi container. Un approccio può essere quello di andare a gestire le iptables presenti nel sistema operativo in esecuzione sulla macchina host e che permettono la creazione e la configurazione di tabelle utili alla gestione dell'inoltro dei pacchetti attraverso le varie interfacce di rete e verso determinate destinazione locali o remate, le quali sono identificate da specifici indirizzi IP e porte. Come citato nell'articolo [29] è possibile configurare nuove regole all'interno delle iptables ed assegnare ad ognuna di queste regole un contesto SELinux anzichè utilizzare gli attributi standard di ACCEPT o DROP per consentire o negare il trasferimento di specifici pacchetti tra un indirizzo sorgente ed uno di destinazione. Di seguito viene indicato un esempio di utilizzo classico di alcune regole di gestione dell'inoltro dei pacchetti.

```
sudo iptables -A FORWARD -p icmp -j DROP
sudo iptables -I FORWARD -p icmp -s 172.17.0.3 -d 172.17.0.5 -j ACCEPT
sudo iptables -I FORWARD -p icmp -s 172.17.0.5 -d 172.17.0.3 -j ACCEPT
```

Questi comandi andranno a popolare le iptables e imporranno di vietare l'inoltro di tutti i pacchetti icmp eccetto quelli provenienti dagli indirizzi 172.17.0.3 e destinati verso l'indirizzo 172.17.0.3 e viceversa. Invece, per poter utilizzare un contesto SELinux al posto dell'attributo DROP o ACCEPT e necessario assegnare le regole di inoltro dei pacchetti all'interno di specifiche tabelle, le quali sono indicate di seguito.

```
mangle
security
```

Queste tabelle permettono di utilizzare delle regole per l'inoltro dei pacchetti includendo l'utilizzo dell'attributo SECMARK, il quale permette di assegnare dei contesti SELinux ad ogni pacchetto di rete in transito da una specifica sorgente e indirizzato verso una specifica destinazione.

SECMARK infatti, è un tool che permette di controllare la rete tramite policy SELinux [30], ma questo strumento non può essere utilizzato per gestire la sicurezza tra sistemi remoti [31]. Un altro componente analogo a SECMARK è Netlabel [32].

Un esempio applicativo è descritto di seguito.

```
sudo iptables -A FORWARD -t mangle -p tcp -s 172.18.0.4 -d 172.18.0.3 -j
SECMARK --selctx system_u:object_r:app1_packet_t:s0
sudo iptables -A FORWARD -t mangle -p icmp -s 172.18.0.3 -d 172.18.0.4 -j
SECMARK --selctx system_u:object_r:app2_packet_t:s0
```

Il contesto SELinux definito nella regola di cui sopra, è costituito da delle etichette SELinux type definite come app1_packet_t e app2_packet_t, in cui la prima etichetta viene assegnata ai pacchetti tcp in uscita dall'indirizzo 172.18.0.4 e destinati verso l'indirizzo 172.18.0.3. La seconda etichetta invece, viene assegnata ai pacchetti icmp in uscita dall'indirizzo 172.18.0.3 e destinati all'indirizzo 172.18.0.4 Queste etichette vengono definite a priori all'interno di una specifica policy SELinux che potrà essere generata in modo automatico o manualmente dall'amministratore del sistema. Un esempio di definizione di tale policy è decritto di seguito.

```
policy_module(secmark, 1.0)
#Type Definitions
type application1_t;
type application2_t;

#Type Packet and Rules Definitions
type app1_packet_t;
corenet_packet(app1_packet_t);
allow application1_t app2_packet_t:packet { recv send };
type app2_packet_t;
dontaudit application2_t app2_packet_t:packet { send };
```

Questo pezzo di codice viene inserito all'interno di un file che dove avere lo stesso nome che è stato definito all'interno della riga policy_module(secmark,1.0).

```
secmark.te
```

La policy definita di cui sopra verrà generata dai seguenti comandi:

```
sudo make secmark.pp
sudo semodule -i secmark.pp
```

Inoltre sarà possibile verificare l'avvenuto caricamento di tale policy con il comando:

```
sudo semodule -l
```

Questo comando infatti permette di visualizzare tutte le policy SELinux attualmente in esecuzione sul sistema host.

Dopo la creazione e l'avvio di tale policy, questà imporrà al processo in esecuzione con l'etichetta SELinux type application1_t di accettare i pacchetti in ingresso e in arrivo etichettati come app2_packet_t. Invece il processo etichettato come application2_t sarà abilitato solo ad inviare i pacchetti etichettati come app2_packet_t, ma non a riceverli. Inoltre,i pacchetti vengono etichettati attraverso le regole precedentemente descritte e che andranno inserite all'interno della iptables.

In seguito sarà inoltre possibile rimuovere tale policy dalla lista di quelle in esecuzione utilizzando il comando:

```
sudo semodule -r secmark
```

Allo stesso modo di come vi è la possibilità di assegnare un contesto ai pacchetti in transito attraverso la rete, è possibile inoltre assegnare dei contesti SELinux anche ai nodi, alle interfacce ed alle porte di rete, attraverso il comando *semanage* [34]. Nel caso di una porta di rete è possibile l'assegnazione di un contesto selinux attraverso il comando indicato di seguito.

```
semanage port -a -t reserved_port_t -p udp 1234
```

L'etichetta reserved_port_t viene definita a priori attraverso delle specifiche policy. Analogamente si può fare anche per le interfacce di rete, dove nell'esempio di seguito, all'interfaccia eth2 viene assegnata un'ettichetta selinux del tipo netif_t.

```
semanage interface -a -t netif_t eth2
```

Infine anche per i nodi di rete, può essere assegnata un'etichetta specifica sempre attraverso il comando *semanage*.

```
semanage node -a -t node_t -p ipv4 -M 255.255.255.255 127.0.0.2
```

3.1.10 SELinux e Networking con interazione fra host differenti

Nel caso in cui fosse necessaria la gestione della sicurezza attraverso SELinux in un contesto basato su applicazioni in esecuzione su macchine differenti le quali comunicano attraverso interfacce di rete, sia in locale che in remoto, è possibile utilizzare degli strumenti che permettono l'inoltro dei pacchetti con l'aggiunta di un nuovo campo all'interno dell'header di tale pacchetto. Tale campo definisce a seconda dello strumento utilizzato, il contesto SELinux assegnato a tale pacchetto, o il solo livello o più precisamente il quarto campo definito come level di un contesto SELinux. Nel primo caso l'etichetta di contesto SELinux verrebbe propagata durante la fase di negoziazione delle *Internet Key*(IKE) per IPsec.

> "Labels propagated during IKE negotiation. Security labels protected by IKE phase-2 encryption. Both peers agree on labels before sending data. Labels assigned to Security Associations (SA). Traffic is implicitly labeled based on matching SA. " [51].

Questo porterebbe ad un maggior overhead dovuto all'inclusione dell'intero contesto SELinux. Nel secondo caso invece si avrebbe un overhead minore ma le funzionalità SELinux tra differenti host risulterebbero limitate a differenza del primo caso. Come descritto negli articoli [51] e [52], gli strumenti che supportano il *Labeled Networking* sono Labeled IPSec [33] e *NetLabel* [32].

Quindi, strumenti come Netlabel e Labeled-IPsec, permettono di gestire la comunicazione di applicazioni in esecuzione su macchine differenti attraverso l'applicazione di policy SELinux. Il primo strumento Netlabel, fornisce il supporto ai Linux Security Modules(LSM), supportando inoltre protocolli per il labeled networking. Le policy LSM vengono quindi estese attraverso la rete in modo tale che il traffico sia soggetto a policy di sicurezza. Con il solo istradamento di una parte del contesto SELinux, ovvero l'MCS/MLS 3.1.8, 3.1.8 e quindi solo il quarto campo di un contesto SELinux, le funzionalità SELinux non risultano essere sfruttate a pieno nell'ambito della gestione dei pacchetti tra host differenti,poichè gli attributi user/role/type sono fissi, sebbene questo garantisca però un minor overhead dei pacchetti in transito. Labeled-IPsec va invece a sfruttare le protezioni IPsec già esistenti come autenticazione e cifratura, oltre a permette la trasmissione completa dell'intero contesto SELinux, ovvero user/role/type/level. Questo però comporta un maggior overhead e quindi una riduzione della prestazioni. Va inoltre precisato che Labeled IPsec può funzionare solo tra sistemi che supportano SELinux. Al fine di poter comprendere con maggiore chiarezza il funzionamento di questo secondo strumento di sicurezza risulta necessario introdurre il concetto di IPsec.

IPsec risulta essere un'architettura di sicurezza a livello IP. Essa è infatti composta da un insieme predefinito di protocolli, oltre alla presenza di ulteriori elementi che descriverò in seguito. I protocolli che vanno a costituire l'architettura di IPsec sono principalmente tre, tra i quali abbiamo: Autentication Header (AH), responsabile di fornire servizi quali autenticazione e integrità

Encapsulation Security Payload (ESP), che serve per garantire servizi quali la riservatezza, l'autenticazione e l'integrità. Internet Key Exchange (IKE), che si occupa della gestione dello scambio di chiavi tra i vari interlocutori. A differenza del protocollo IKE, i primi due protocolli, non si curano dello scambio delle chiavi, ma prendono per buono che tale negoziazione sia andata a buon fine ed è stata creata una Security Association (SA), ovvero una convenzione che ne descrive al suo interno i meccanismi di sicurezza che andranno utilizzati e con l'ausilio di quali chiavi. Quindi sia AH che ESP possono essere utilizzati in modalità trasporto o in modalità tunnel. Bisogna precisare inoltre che la modalità trasporto non è utilizzabile tra i security gateway(). Gli header di questi due protocolli sono inseriti tra l'header IP e l'header dei protocolli TCP o UDP, a differenza che in modalità tunnel, il pacchetto di origine, viene incapsulato in un nuovo pacchetto IP il quale avrà un nuovo header, come illustrato nella figura 3.8

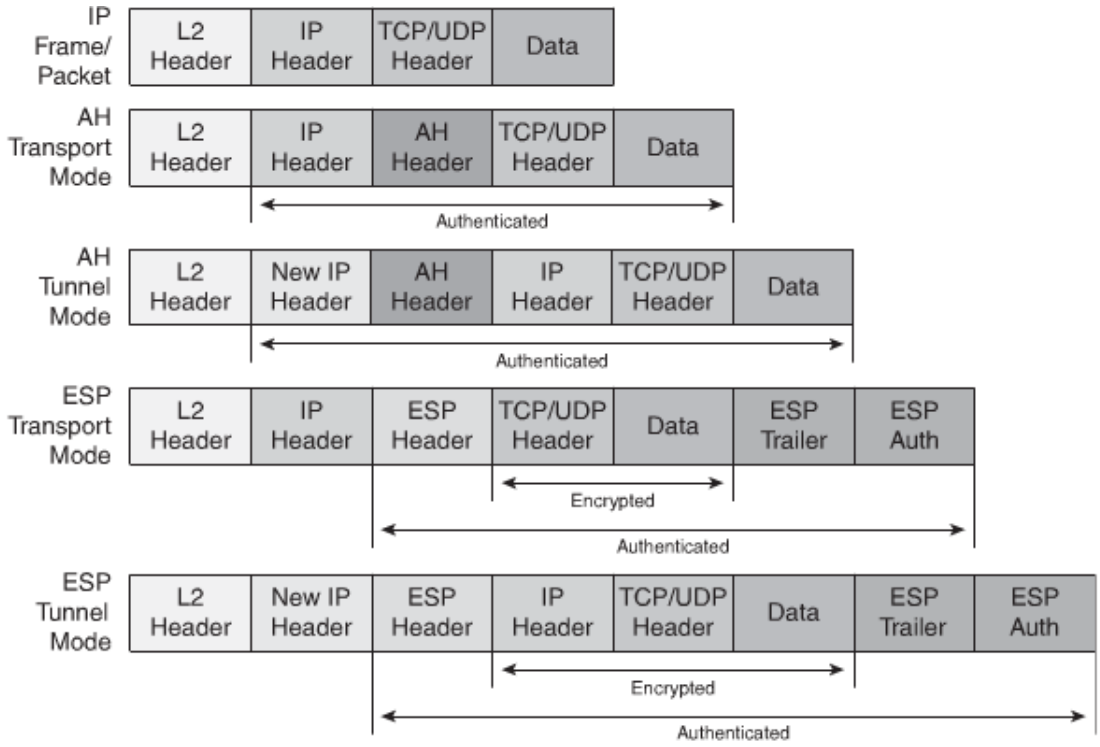

Figura 3.8. IPsec AH and ESP [53].

Invece, come anticipato in precedenza, il concetto di Security Association definisce una relazione tra due o più entità, andando a descrivere quindi come le stesse entità useranno i servizi di sicurezza allo scopo di poter effettuare una comunicazione sicura. Quindi IPSec offre molte opzioni per l'esecuzione della crittografia e dell'autenticazione di rete. Infatti, ogni connessione IPSec, come detto in precedenza può fornire crittografia, integrità, autenticità o tutti e tre. All'atto della determinazione del servizio di sicurezza, i due peer IPSec dovranno quindi determinare esattamente quali algoritmi utilizzare, come ad esempio, DES o 3DES per la crittografia, MD5 o SHA per l'integrità. Dopo aver deciso gli algoritmi, i dispositivi si troveranno a dover condividere le chiavi di sessione. Quindi la security Association è in fin dei conti un metodo che viene utilizzato da IPSec allo scopo di poter tenere traccia di tutti i dettagli relativi a una determinata sessione di comunicazione IPSec, ovvero il traffico che vi fluisce all'interno di essa. Da notare inoltre che le SA sono unidirezionali, per cui al fine di poter permettere una comunicazione ad esempio tra due host, saranno necessarie due SA. Inoltre ogni SA non può essere associata in contemporanea ad AH o ESP. Tutte le SA attive all'interno di un host o di un Security Gateway, sono contenute al'interno di un Security Association Database(SAD). Esiste inoltre un ulteriore database che è il Security Policy Database (SPD), il quale contiene tutte le policy di sicurezza utili a comprendere se un determinato pacchetto debba essere o meno scartato, lasciato in chiaro, o elaborato tramite IPSec, il tutto avvalendosi di parametri come l'indirizzo IP sorgente e destinazione, la porta sorgente e o destinazione, e il protocollo di trasporto utilizzato. L'introduzione all'architettura di

IPsec e il suo funzionamento mi è stato utilie per poter descrivere come SELinux gestisce la comunicazione remota o locale tra macchine differenti. Infatti il concetto di gestione della sicurezza attraverso l'utilizzo di Policy SELinux non è più legato ad una singola macchina host sulla quale vengono eseguite più applicazioni in grado di comunicare tra loro utilizzando i vari meccanismi di Inter Process Communication, ma si estende anche alla comunicazione tra processi in esecuzione su macchine differenti.

Dopo l'introduzione dell'architettura IPsec risulta quindi possibile passare ad una descrizione più accurata di Labeled IPsec, come citato nell'articolo [54]. Al fine di poter applicare le opportune etichette a ciascun pacchetto in transito su sistemi differenti, risulta necessario utilizzare uno strumento come ipsec-tool. Bisogna precisare inoltre che poichè questa procedura vada a buon fine è necessario che anche il kernel del sistema operativo supporti il labeling.

Si andrà quindi a considerare un client ed un server installati ciascuno su macchine differenti alle quali sono stati assegnati i seguenti indirizzi ip:

- 192.168.147.132 Server
- 192.168.147.130 Client

Dopo l'avvio del server ed in seguito del clinet, si riceveranno gli output descritti di seguito.

```
[root@poisonivy ~]# ./client 192.168.147.132

getpeercon: Protocol not available Received: Hello, (null) from (null)

[root@scarecrow ~]# ./server

getsockopt: Protocol not available server: got connection from
    192.168.147.130, (null)
```

Il metodo *getpeercon()* ritorna quel tipo di risposta, poichè il labeling non risulta ancora abilitato. Andrà quindi creata un SA tra le 2 macchine, oltre a specificare un contesto SELinux come descritto di seguito.

```
[root@scarecrow ~]# cat dev/ipsec/setkey.scarecrow.test

spdflush;

flush;

spdadd 192.168.147.130 192.168.147.132 any

   -ctx 1 1 "system_u:object_r:default_t:s0"

   -P in ipsec esp/transport//require;

spdadd 192.168.147.132 192.168.147.130 any

   -ctx 1 1 "system_u:object_r:default_t:s0"

   -P out ipsec esp/transport//require;
```

Nell' esempio di cui sopra si evince come prendendo in considerazione la parte lato server, sia stato possibile aggiungere un contesto tramite l'utilizzo dell'attributo -ctx all'interno di un file configurazione gestibile attraverso il comando *setkey*. Tale comando permette di poter modificare manualmente le Security Association e le Security Policy che si andranno a creare all'interno dei loro corrispettivi database quali SAD e SPD[55].

Attraverso l'utilizzo del comando indicato di seguito, sarà possibile avviare il cliente con un contesto differente

```
[root@poisonivy ~]# runcon -t httpd_t ./client 192.168.147.132

Received: Hello, root:system_r:httpd_t:SystemLow-SystemHigh from
    root:system_r:unconfined_t:-SystemHigh
```

Si nota come il server veda l'etichetta di contesto SELinux settata come httpd_t lato client all'atto della chiamata del metodo getpeercon(). Questo quindi ci permette di identificare il contesto del processo in questione relativamente all'altro lato della connessione. Adesso prendiamo in considerazione le policy che si andranno ad applicare allo scopo di poter gestire la comunicazione tra le due macchine. Attraverso l'utilizzo della policy seguente, si andrà ad impostare manualmente un regola che permette al processo client in esecuzione con l'etichetta di contesto http_t di applicare una corrispondenza di httpd_t al contesto precedentemente definito all'interno del *Security Policy Database*(SPD). Questo permette di poter applicare dunque, una policy SELinux ad uno specifico dominio. Un esempio può essere quello di considerare due domini ben distinti nei quali, uno di questi utilizzerà una crittografia avanzata, e l'altro dominio consentirà ad altri utenti l'utilizzo di una crittografia più veloce ma di qualità minore, il tutto allo scopo di poter specificare con quali tipi di SPD un dominio possa essere messo a confronto.

```
allow httpd_t default_t:association polmatch;
```

La policy indicata di seguito, indica che si andrà ad inviare attraverso la SA etichettata come httpd_t.

```
allow httpd_t self:association sendto;
```

Questo tipo di controllo descritto di seguito risulta invece molto significativo, poichè si va a consentire ad un dominio etichettato come httpd_t di ricevere pacchetti attraverso una SA etichettata come unconfined_t. Questo tipo di policy permette quindi di utilizzare degli specifici criteri utili a poter determinare quali domini presenti su un'altra macchina possano inviare dati ad altri domini (da definire quali) presenti su un'altra macchina.

```
allow httpd_t unconfined_t:association recvfrom;
```

Con quest'ultima policy indicata di seguito, si va a definire il dominio associato alla entry nella SPD che è stata associata per un dominio remoto.

```
allow unconfined_t default_t:association polmatch;
```

Prima di proseguire con la descrizione di tale policy, si andrà a descrivere in breve come sia necessario circoscrivere i domini in locali e remoti.

Quindi, in riferimento a ciascun lato della connessione vengano definite 2 SA, di cui una in entrata ed una in uscita e entrambe marcate con appropriate etichette di contesto SELinux. Bisogna precisare inoltre che le SA definite come uscenti all'interno di ogni sistema devono essere circoscritte all'interno di un domino locale. Per quanto riguarda le SA in entrata che sono state definite per ogni sistema, dovranno essere circoscritte all'interno di un dominio remoto [54]. Tornando alla descrizione dell'ultima policy, risulta più chiaro comprendere per quale motivo siano state definite due voci SPD, ovvero una per gestire le comunicazioni in uscita e una per le comunicazioni in entrata. Inoltre, entrambi i domini dovranno eseguire la corrispondenza degli SPD sulla macchina locale (per le comunicazioni in uscita) e sulla macchina remota (per la ricezione). Questo meccanismo di labeled ipsec permette quindi di definire dei criteri da assegnare alla macchina sorgente e a quella destinazione allo scopo di poter effettuare comunicazioni crittografate e sicure, e all'interno di domini definiti da specifici contesti SELinux. Questo tipo di soluzione è molto simile ai meccanismi di Inter Process Communication utilizzati in combinazione con SELinux, ma in questo caso la comunicazione non avviene più sulla medesima macchina, ma su macchine ben distinte.

3.1.11 Differenze tra Seccomp, SELinux e AppArmor

Di seguito metterò a confronto le funzionalità di Seccomp rispetto agli strumenti dedicati al Mandatory Access Control quali SELinux e Apparmor, allo scopo di poter fornire una comprensione

sul fatto che tali strumenti possano o meno coesistere nello stesso sistema e in che modo oppure, in caso contrario, quali di essi risultano escludersi a vicenda. Tutti e tre gli strumenti, si basano sugli stessi meccanismi di intercettazione o filtraggio delle syscall a livello kernel, ed esse sono guidate da delle policy applicabili per ogni singolo processo, ma vi sono comunque delle differenze che verranno descritte di seguito.

Quindi i linguaggi per le policy, utilizzati da AppArmor e SELinux si differenziano tra loro per quanto riguarda la semplicità di utilizzo degli stessi e la terminologia specifica, risultando quindi più complessi e con maggiori possibilità di personalizzazione delle policy, rispetto a Seccomp. Sia SELinux che Apparmor, consentono la definizione di attori che generalmente sono dei processi, da azioni, come ad esempio la lettura di file, le operazioni di rete ecc.., e dalla definizione di obiettivi associati ad asempio a dei file, indirizzi IP, protocolli e porte, a differenza di Seccomp, che utilizza un semplice elenco di chiamate di sistema all'interno di una blacklist(tutte le syscall non presenti all'itenro di tale lista saranno permesse) o all'interno di una whitelist(solo le syscall presenti nella lista saranno concesse). Inoltre tutte le chiamate di sistema non concesse da Seccomp, porteranno ad un arresto del porcesso, a differenze degli altri due strumenti(SELinux e Apparmor) che non arrestano il processo, ma bloccano solo la chiamata non concessa, mostrando un avviso e memorizzando un reporto relativo ai tentativi di eseguire delle syscall non concesse dalle policy in esecuzione sul sistema. Un altra differenza consiste nel fatto che Seccomp è volontario, ovvero i processi possono passare volontariamente ad uno stato limitato chiamando prctl (PR_SET_SECCOPMP, ...) a differenza dei Linux security modules, i quali, essendo strumenti ad accesso obbligatorio, sono costituiti da dei criteri che vengono definiti e caricati prima dell'esecuzione di uno specifico processo. Inoltre, poichè Seccomp è stato incluso nei kernel di Linux dalla v2.6.12, non richiede una configurazione esplicita come SELinux. Uno strumento utile a verificare quali syscall sono abilitate per ciascuna etichetta di contesto assegnata da SELinux è sesearch. Di seguito ho indicato un esempio pratico che ne descrive il funzionamento.

```
sesearch -t netutils_t -c capability -Ad

#OUTPUT:
Found 1 semantic av rules:
allow netutils_t netutils_t : capability {chown dac_read_search
setgid setuid net_admin net_raw sys_chroot }
```

In questo caso vengono mostrate le syscall associate al contesto netutils_t. E' quindi importante evidenziare come la coesistenza di SELinux e Apparmor non sia possibile in quanto entrambi sono due strumenti di Mandatory Access Control. Va ora analizzata una possibile coesistenza tra uno dei due Linux Security Module e Seccomp. Facendo Riferimento a SELinux, questo permette una gestione delle syscall, con una precisione maggiore rispetto a Seccomp, infatti le syscall vengono associate a ciascun contesto del dominio associato a tale processo, un dominio permette di definire a quali file, directory, collegamenti, dispositivi o porte un dato processo può accedere, associando a ciascuno di essi un'etichetta di contesto. Se prendiamo in considerazione un processo che esegue un server Apache (Httpd), potremmo volere per determinati motivi che tale processo abbia abilitate le syscall di read e di write abilitate sui file presenti in una directory ad esempio folder1 e la syscall di read abilitata per tutti i file della directory folder2. In questo caso ai file della folder1 sarà possibile accedervi sia in lettura che in scrittura, a differenza dei file presenti nella folder2 i quali sarà possibile solo eseguirli in lettura. Questa procedura invece con Seccomp non sarebbe possibile, poichè Seccomp permette l'abilitazione delle syscall per processo e non per contesto associato alle classi (directory,ports,files) il cui dominio è lo stesso del processo in questione, quindi un processo con syscall read write abilitata, potrà eseguire la lettura e scrittura di tutti i file presenti in qualunque direcotry ad esso associata. Questo perche con Seccomp, come descritto nella sezione 3.1.5 le capabilities vengono gestite per processo e quindi si avrà una lista che potrà essere standard o personalizzata e che a sua volta potrà essere definita come una blacklist o una whitelist. In conclusione, entrambi SELinux e Seccomp permettono di gestire le capabilities a livello di singolo processo, ma con la differenza che con SELinux è possibile una gestione delle syscall ad un livello di granularità maggiore, grazie alla possibilità di abilitarle per contensto. Dopo questa analisi ritengo che Seccomp possa essere escluso se si utilizzano strumenti di Mandatory Access Control, poichè essi forniscono di base le stesse funzionalità, ma in aggiunta

permettono anche se ad un livello di complessità maggiore, delle configurazioni più specifiche delle syscall a livello di contesto e non di processo, tramite la definizione di moduli di policy.

3.1.12 Container bridge networking

I container stanno cambiando il modo in cui le applicazioni vengono sviluppate e come le applicazioni si interfacciano con la rete. La rete diventa quindi critica per le implementazioni di produzione perché la rete ha bisogno di essere automatizzata, scalabile e sicura, per adattarsi alla nuova generazione di architetture di cloud ibridi e delle applicazioni basate su microservices.

Ci sono diverse opzioni disponibili per i container di rete e gli sforzi di standardizzazione hanno cominciato a succedere sui diversi approcci. Prima di esaminare i diversi standard esistenti, esaminiamo le interfacce di rete per i container, confrontandole con quelle delle macchine virtuali.

Le macchine virtuali simulano l'hardware e includono le schede di interfaccia di rete virtuali (NIC) utilizzate per la connessione alla NIC fisica. D'altra parte, i container sono solo processi, gestiti da un container di runtime, che condividono lo stesso kernel del sistema operativo presente nella macchina sottostante. Quindi i container possono essere collegati alla stessa interfaccia di rete e namespace 3.1.1 di rete dell'host (ad esempio, eth0), oppure possono essere collegati a un'interfaccia di rete virtuale interna e con proprio namespace di rete e quindi collegati al mondo esterno in vari modi.

Gli schemi iniziali di rete dei container si occupano di come collegare gli stessi, in esecuzione su un singolo host e renderli raggiungibili dalla rete. Nella modalità 'host', i container vengono eseguiti nello spazio dei nomi di rete host e utilizzano l'indirizzo IP della macchina host. Per esporre il container all'esterno dell'host, esso dovrà utilizzare una porta dallo spazio dell'host. Ciò significa che è necessario gestire le porte che i container contengono, poiché tutti condividono lo stesso spazio di porte.

La modalità "bridge" invece, offre un miglioramento rispetto alla modalità "host" . Nella modalità "bridge" , i container ottengono gli indirizzi IP da una rete privata e vengono collocati nei rispettivi spazi dei nomi di rete. Poiché i container sono nel proprio spazio dei nomi, hanno anche il proprio spazio di porte e non devono preoccuparsi dei conflitti tra le porte. Ma i container risultano ancora esposti all'esterno dell'host, se utilizzando l'indirizzo IP dell'host. Ciò richiede l'uso di NAT (traduzione di indirizzi di rete) per mappare tra IP host: porta host e IP privato: porta privata. Queste regole NAT vengono implementate utilizzando le Linux iptables, che però vanno a limitare le prestazioni. Il NAT viene utilizzato per fornire comunicazione oltre l'host. Mentre le reti bridge risolvono i problemi di conflitto di porta e forniscono l'isolamento di rete ai container in esecuzione su un host. Vi è però un costo di prestazioni relativo all'utilizzo del NAT. Ecco un esempio del flusso di creazione.

1)Come primo passo viene creato un bridge all' interno dell'host.

2)In seguito viene assegnato un namespace per ogni container all'interno diel bridge.

3)L'ethX dei container viene mappata come private bridge interface.

4)Le iptables in combinazione con il NAT vengono utilizzate per il mapping tra le interfacce private dei container e l'interfaccia pubblica dell'host.

3.1.13 Procfs e Sysfs

Procf, situato in / proc, è statio originariamente progettato per esportare tutte le informazioni di un processo come ad esempio lo stato corrente di tale processo o tutti i descrittori di file aperti nello spazio utente. Nonostante i suoi scopi iniziali, procfs viene utlizzato molti altri scopi come fornire informazioni sul sistema in esecuzione come; informazioni sulla CPU, informazioni sugli interrupt,

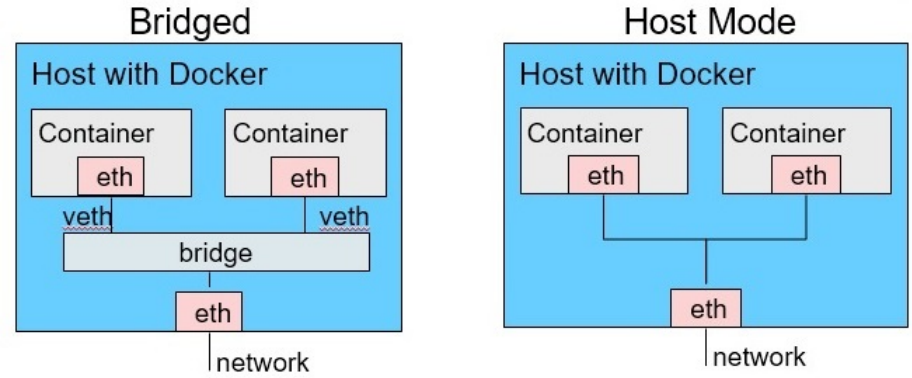

Figura 3.9. Bridged vs Host Mode [35].

sulla memoria disponibile o sulla versione del kernel, fornire informazioni su “ide devices” , “scsi devices” , informazioni di rete come la tabella arp, le statistiche di rete o gli elenchi di socket utilizzati. Esiste inoltre una sotto directory speciale: / proc / sys, che consente di configurare molti parametri del sistema che è in esecuzione. Di solito ogni file è costituito da un singolo valore, e questo valore può corrispondere ad un limite, come ad esempio dimensione massima del buffer, può servire per attivare o disattivare una determinata funzionalità come ad esempio il routing, o per rappresentare un’altra variabile del kernel. Tutte le directory ed i file sotto / proc / sys / non vengono implementati con l’interfaccia procfs, che invece usano un meccanismo chiamato sysctl, che è uno strumento utile per esaminare e modificare i parametri del kernel in fase di runtime.

Da notare che nonostante l’ampio utilizzo dei procfs, esso è stato deprecato per essere utilizzato solo per esportare informazioni relative a un processo stesso. Per utilizzare procf è necessario che esso sia compilato con il codice sorgente del kernel di Linux. Questo viene fatto impostando il parametro CONFIG_PROC_FS = y. Bisogna tener presente che nella maggior parte delle configurazioni standard è abilitato per impostazione predefinita. Procf supporta inoltre due diverse API per i moduli del kernel: L’API legacy procfs, che risulta molto facile da usare finché la quantità di dati da gestire è minima, ovvero si considera come minima, meno di una dimensione di pagina (PAGE_SIZE), che nei sistemi i386 è di 4096 byte. L’API Seq_file, dove Seq_file è stato progettato per facilitare la gestione delle richieste di lettura. Questa API supporta richieste di lettura per più di PAGE_SIZE byte e fornisce un meccanismo per attraversare un elenco, raccogliere gli elementi dell’elenco e inviare tutti gli elementi allo spazio utente. Come ho anticipato in precedenza, l’API legacy procfs consente la creazione di file e directory, e per ogni file è necessario specificare due funzioni di richiamata, di cui una viene eseguita quando un utente legge il file e l’altra quando un utente scrive sul file. L’utilizzo di questa API è ben descritto nella “Linux Kernel Procfs Guide” distribuita con il codice sorgente del kernel Linux.

Sysfs invece è stato progettato per rappresentare l’intero modello del dispositivo visto dal kernel Linux. Questo infatti contiene informazioni sui dispositivi, sui driver, sui bus e sulle loro interconnessioni. Sysfs è inoltre fortemente strutturato per rappresentare la gerarchia e le interconnessioni, e contiene molti collegamenti tra le singole directory. Per quanto riguarda il kernel, esso contiene le seguenti directory di primo livello:
sys / block: tutti i dispositivi di blocco noti come hda / ram / sda /
sys / bus: tutti gli autobus registrati. Ogni directory sotto bus / contiene in modo predefinito due sottodirectory, ovvero quella di device / per tutti i dispositivi collegati a quel bus, e quella di driver / per tutti i driver assegnati con quel bus.
sys / class: dove per ogni tipo di dispositivo esiste una sottodirectory: ad esempio / stampante o / sound.
sys / device: contiene tutti i dispositivi noti al kernel, che sono organizzati dal bus a cui sono connessi.
sys / firmware: in questa directory sono presenti i firmware di alcuni dispositivi hardware.
sys / fs: directory usata per controllare un file system attualmente utilizzato da FUSE, un’implementazione del file system di spazio utente.

sys / kernel: tiene le directory o cosiddetti “mount points” per altri filesystem come debugfs, securityfs.
sys / module: dove ogni modulo kernel caricato viene rappresentato con una directory.
sys / power: file per gestire lo stato di alimentazione di alcuni componenti hardware.

Al fine di poter utilizzare sysfs, esso deve essere compilato con il codice sorgente del kernel Linux. Questo viene fatto impostando il parametro CONFIG_SYSFS = y. La filosofia dietro sysfs è quella di rappresentare ogni valore con un file dedicato. Inoltre ogni file ha una dimensione massima di byte PAGE_SIZE. Per un modulo kernel ci sono tre possibilità per utilizzare un file sotto / sys e sono;

parametro del modulo
registrazione del nuovo sottosistema
debugfs, che è montato in / sys / kernel / debug.

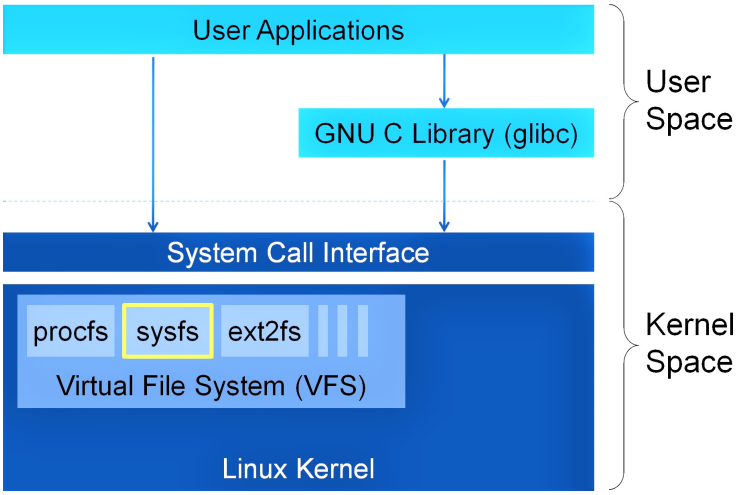

Figura 3.10. Sysfs e Procfs [36].

3.1.14 Root Capability Dropping

Al fine di eseguire controlli di autorizzazione,le implementazioni UNIX tradizionali, distinguono due categorie di processi tra cui quelli privilegiati, ovvero quei processi il cui ID utente effettivo è 0, indicato come superutente o root, e i processi non privilegiati, il cui UID effettivo non è nullo. I processi privilegiati escludono tutti i controlli di autorizzazione del kernel, mentre quelli non privilegiati sono soggetti al controllo completo delle autorizzazioni sulle credenziali del processo. A partire dal kernel 2.2, Linux divide tradizionalmente i privilegi associati all’utente superuser in unità distinte, conosciute come capabilities, che possono essere abilitate e o disabilitate ??in modo indipendente. Le capabilities sono assegnate per singolo thread o processo.
Come detto in precedenza, le capabilities in Linux servono per gestire i privilegi disponibili per i processi eseguiti come utente root e ridurli in gruppi di privilegi più piccoli. In questo modo un processo in esecuzione con privilegio di root può essere limitato ad ottenere solo le autorizzazioni minime necessarie per eseguire il suo funzionamento. I container engine, supportano le funzionalità Linux come parte del comando di esecuzione. Se ad esempio prendiamo in considerazione Docker avremo i seguenti comandi: “cap-add” e “cap-drop” . Per impostazione predefinita, viene avviato un container con diverse funzionalità consentite per impostazione di default e tali impostazioni possono essere successivamente eliminate o rimosse se non necessarie. Ulteriori permessi in aggiaunta a quelli di default, possono essere aggiunti manualmente. Entrambi i comandi ”cap-add” e ”cap-drop” supportano il valore ALL, che consente di rimuovere tutte le funzionalità assegnate ad un dato processo. E’ importante tenere presente che le funzionalità minime necessarie dipendono dalle applicazioni, e la definizione di queste funzioni può richiedere diverso

tempo e l'esecuzione di diversi test. Un importante accorgimento è quello di non utilizzare mai la funzionalità SYS_ADMIN a meno che non sia specificamente richiesto dall'applicazione. Sebbene le funzionalità abbattono le potenzialità di root in piccoli frammenti, SYS_ADMIN di suo concede una gran parte delle funzionalità, mettendo a disposizione una maggiore superficie di attacco.

Di seguito ho elencato tutte le funzionalità abilitate per impostazione di default nel caso in cui viene eseguito un container, elencando inoltre le loro descrizioni.

CHOWN - Effettua modifiche arbitrarie ai file UID e GID.

DAC_OVERRIDE - Effettua il controllo dell'accesso discrezionale (DAC), bypassando i file in lettura, scrittura ed eseguendo il controllo dei permessi.

FSETID - Non cancella i bit di modalità set-user-ID e set-group-ID quando un file viene modificato, inoltre setta il bit di set-group-ID per un file il cui GID non corrisponde al file system o ad uno qualsiasi dei GID aggiuntivi del processo di chiamata.

FOWNER - Effettua il controllo delle autorizzazioni di bypass per le operazioni che normalmente richiedono che l'UID del file system del processo corrisponda all'UID del file, escludendo le operazioni coperte da CAP_DAC_OVERRIDE e CAP_DAC_READ_SEARCH.

MKNOD - Crea file speciali usando mknod.

NET_RAW - Utilizza sockets RAW e PACKET, al fine di eseguire il binding a qualsiasi indirizzo per un "proxy trasparente" , al fine di intercettare la normale comunicazione al network layer, senza richiedere alcuna configurazione particolare del client. I client infatti non devono essere consapevoli dell'esistenza di tale proxy. Di norma si trova un proxy trasparente tra il client e Internet, con il proxy che esegue alcune delle funzioni di un gateway o di un router.

SETGID - Esegue manipolazioni arbitrarie del GID di un processo.

SETUID - Realizza manipolazioni arbitrarie dell' UID di processo.

SETFCAP - Imposta le funzionalità dei file.

SETPCAP - Viene utilizzato se le funzionalità di un file non sono supportate, e permette di concedere o rimuovere qualunque capabilities all'interno della funzionalità del chiamante che è stata impostata da o verso qualunque altro processo.

NET_BIND_SERVICE - Permette di collegare un socket a porte privilegiate del dominio Internet, con numeri di porta minori di 1024.

SYS_CHROOT - Permette l'utilizzo di chroot per passare a una directory radice differente.

KILL - Bypassa il controllo delle autorizzazioni per l'invio di segnali.

AUDIT_WRITE - Permette di scrivere i record nel registro di controllo del kernel.

Generalmente per la maggior parte delle applicazioni eseguite all'interno dei container, è possibile disattivare da questa lista predefinita; AUDIT_WRITE, MKNOD, SETFCAP, SETPCAP, perchè non strettamente necessarie.

3.1.15 Sysdig, Csysdig e Falco

Sysdig è uno strumento che fornisce funzionalità utili per eseguire una profonda analisi del sistema, e che in aggiunta ha un supporto nativo per i container.
Sysdig è stato sviluppato allo scopo di consentire un accesso facilitato al comportamento effettivo dei sistemi e dei container Linux.

Solitamente, il monitoraggio e la risoluzione dei problemi a livello di sistema implicano ancora l'accesso a una macchina con SSH e l'utilizzo di un vasto numero di strumenti datati e con interfacce molto incoerenti. Inoltre molti di questi classici strumenti di Linux non risultano essere disponibili negli ambienti di container. Sysdig unisce il tuo toolkit Linux in un'unica interfaccia che allo stesso tempo offre coerenza e facilità di utilizzo. L'architettura unica di sysdig permette dunque un'ispezione profonda nei container, rimanendo esterna ai container stessi. Questo strumento Sysdig offre come valore aggiunto la possibilità di concentrarsi su alcuni principi fondamentali, offrendo supporto nativo per tutte le tecnologie di gestione dei container Linux, tra cui Docker, LXC, LXD, e Rocket, offrendo una visibilità unificata, coerente e granulare nei sottosistemi di memorizzazione, elaborazione, rete e memoria, e consentendo di creare file di traccia per l'attività del sistema, analogamente a quanto si può fare per reti con strumenti come tcpdump e Wireshark, in modo che il problema possa essere analizzato in un secondo momento senza dover necessariamente perdere informazioni importanti. Inoltre l'attività catturata può essere posta in pieno contesto offrendo un linguaggio filtrante utile per permettere di scavare a fondo e trovare le informazioni in modo naturale e interattivo.
Sysdig si installa sulle macchine fisiche e/o virtuali a livello di sistema operativo, all'interno del kernel Linux per poter catturare le chiamate di sistema ed altri eventi di sistema. Quindi, tramite l'utilizzo dell'interfaccia da riga di comando di sysdig o dell'UI curses-based, è possibile filtrare e decodificare questi eventi per estrarre informazioni utili. Sysdig può essere inoltre utilizzato per controllare i sistemi in real time, o per generare file di traccia che possono essere analizzati in una fase successiva. Inoltre fornisce anche una ricca libreria di script utili per risolvere i problemi più comuni, offrendo un'interfaccia UI curses-based semplice e completamente personalizzabile che prende il nome di csysdig.
Csysdig sfrutta il sistema di raccolta sysdig, esportando in un'interfaccia utente che è possibile immaginare come una combinazione di molti strumenti di monitoraggio della riga di comando popolari, strumenti come strace, tcpdump, htop, iftop e lsof.
Proprio come il sysdig, Csysdig supporta nativamente i container in modo pulito ed elegante, permettendo l'esecuzione all'nterno del prorpio container o direttamente sulla macchina host sottostante, per poter effettuare un'analisi completa delle vulnerabilità presenti in tutti i container che sono in esecuzione sulla macchina host. Csysdig fa parte del pacchetto sysdig, ed è quindi incluso nativamente ed installato, all'atto dell' installazione sysdig sulla macchina. Inoltre csysdig si basa sul concetto di "visualizzazioni" , ovvero piccoli script di Lua che determinano come vengono raccolte, elaborate e rappresentate le metriche sullo schermo. Includendo una nuova visualizzazione a csysdig non è necessario richiedere l'aggiornamento del programma, ma ci si basa più semplicemente sull'aggiunta di una nuova "vista" . Le viste si basano sul motore di elaborazione sysdig e questo significa che possono includere qualsiasi filtro sysdig. Inoltre le viste si trovano nel percorso della directory di sysdig chisel, di solito in /usr/share/sysdig/chisels/.chisels.
Csysdig è stato progettato per simulare strumenti come top e htop, ma offre anche funzionalità più ricche, tra cui il supporto per analisi dal vivo e file di traccia sysdig. I file di traccia possono provenire dalla stessa macchina o da un'altra macchina. Fornisce la possibilità di analizzare le metriche di CPU, memoria, I/O del disco, I/O di rete. Ha la capacità di osservare l'attività di input/output per i processi, i file e connessioni di rete. Analizza i processi, i file, le connessioni di rete al fine di ottenere un'ananalisi più approfondita sul loro comportamento. Vi è un supporto completo di personalizzazione, e un supporto per il linguaggio di filtraggio di sysdig. Csysdig ha inoltre la possibilità di funzionare su un qualsiasi terminale. Nella figura 3.11 ed in quelle a seguire, presenti come riferimento nel proseguo del descrizione, verrà illustrato un esempio di utilizzo di Csysdig in CentOS 7.

Dall'interfaccia grafica illustrata in figura 3.11, si evince come attraverso la selezione del comando *view* situato in basso a sinistra o più semplicemente digitando F2 da tastiera, vengano mostrate tutte le sezioni analizzabili per l'intera macchina host, o ad esempio per singolo container in esecuzione sulla specifica macchina host. Se ad esempio selezioniamo la parte relativa ai container, verranno mostrate tutte le applicazioni in esecuzione all'interno di ciascun container. In questo caso per un'analisi più approfondita è stato selezionato uno dei container che eseguono l'immagine di un server Apache(httpd) 3.12.

Dalla figura 3.13 si vede come nella parte alta della finestra, dopo la dicitura for: non sia più indicato *whole machine*3.12, ma container seguito dal suo ID associato.

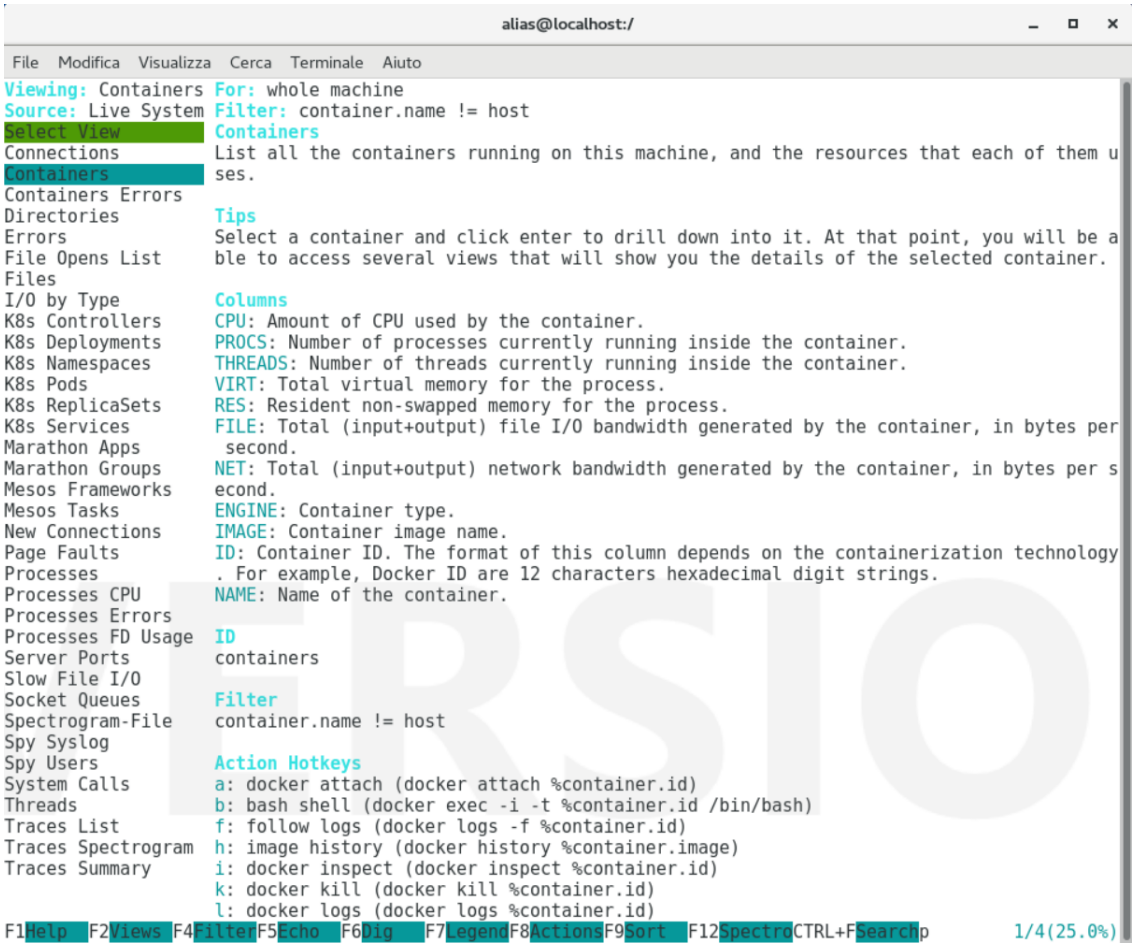

Figura 3.11. Csysdig.

A questo punto come passo successivo, sempre tramite il tasto F2 sarà possibile visualizzare, come mostrato in figura 3.13 la lista delle sezioni analizzabili, ma questa volta associate al singolo container e non all'intera host machine. Allo stesso modo risulterà quidni possibile visualizzare le directory associate al container in questione, i files, le system calls eseguite per tale container oltre a tutte le altre funzionalità elencate.

Nella figura 3.14 vengono mostrate le syscall attualmente utilizzate dal container in questione. Questo strumento, permette quindi di effettuare un'analisi approfondita per ciascun container, analizzando inoltre, gli errori per container, eventuali page fault, oltre alla possibilità di analizzare le funzionalità di rete.

Un altro importante strumento è Sysdig Falco, un monitor di sicurezza open-source dei container, che è stato progettato allo scopo di rilevare attività anomale nelle applicazioni sulla base della tecnologia di monitoraggio di Sysdig. Falco consente di monitorare e rilevare continuamente l'attività di un container, files, applicazioni, syslog, rete, funzionando anche come un sistema di rilevamento delle intrusioni su un qualsiasi host Linux, grazie ad un insieme di regole personalizzabili. Si può pensare a falco come una combinazione di snort, ossec e strace, ma più semplice da utilizzare. Falco è stato estratto da un'importante realizzazione la quale tende a sottolineare che i dati all'interno delle system-call possono essere utilizzati per proteggere le applicazioni, anche all'interno dei container. Infatti al contrario di altri approcci, Falco riesce nel suo intento in modo più semplice, unificato ed accessibile a tutti.
E' importante tenere in considerazione il fatto che Falco è uno strumento di controllo che risulta essere in contrasto con altri strumenti come Seccomp o AppArmor. Esso viene eseguito nello spazio utente, utilizzando un modulo del kernel per recuperare le chiamate di sistema, mentre altri strumenti simili eseguono il filtro/monitoraggio delle chiamate di sistema a livello di kernel.

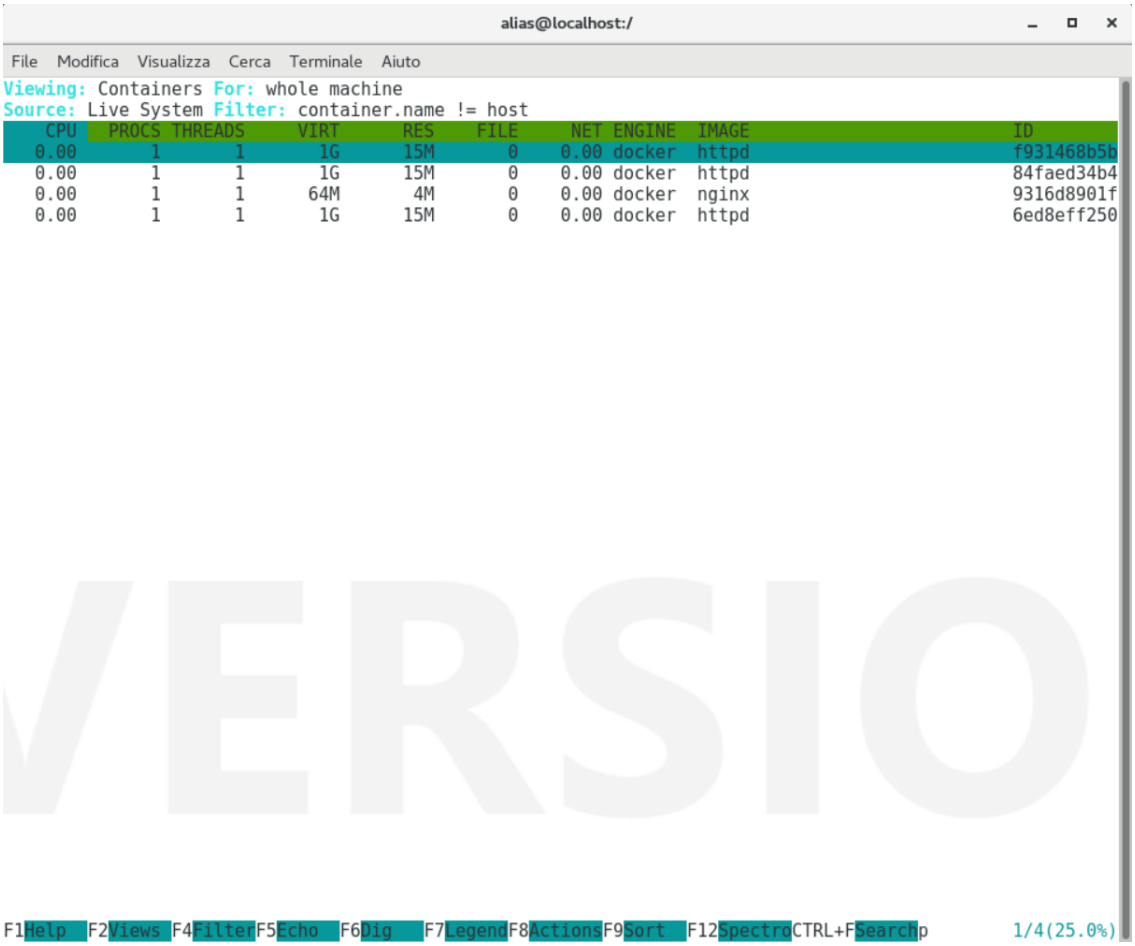

Figura 3.12. Csysdig whole machine.

Uno dei vantaggi di un'installazione dello spazio utente è che si è in grado di integrarsi con sistemi esterni come Docker, Docker Swarm e Kubernetes, con la possibilità di incorporare i metadati e i tag.

3.1.16 Docker Bench Security

Docker bench security è uno costituito da uno script che ha il compito di controllare un certo numero di best-practice che risultano essere comuni nell'ambito di distribuzione dei container in Docker.
Con l'esecuzione dello script presente in questo strumento, vengono visualizzate delle informazioni relative alla corretta configurazione del container management system e accanto ad ognua di esse viene indicato se le best-practies di configurazione sono state implementate in modo corretto o meno, o anche in modo parziale. Questo insieme di best-practies viene raggruppato in 5 sezioni quali configurazione dell'host, configurazione di Docker Daemon,Docker Daemon Configuration Files, immagini di container e file di archiviazione e runtime del container.

Di seguito elencherò le best-practies che si andranno a verificare.

Create a separate partition for container.
Solitamente è consigliabile creare una partizione separate per il docker daemon, che sarà situata in /var/ lib/docker. E' molto importante, fare lo stesso anche per i Docker container, andano a memorizzare quindi i dati sensibili in una partizione separata dal resto. Questo semplifica drasticamente il nostro processo di backup e ripristino. Infatti, quando qualcosa non va a buon

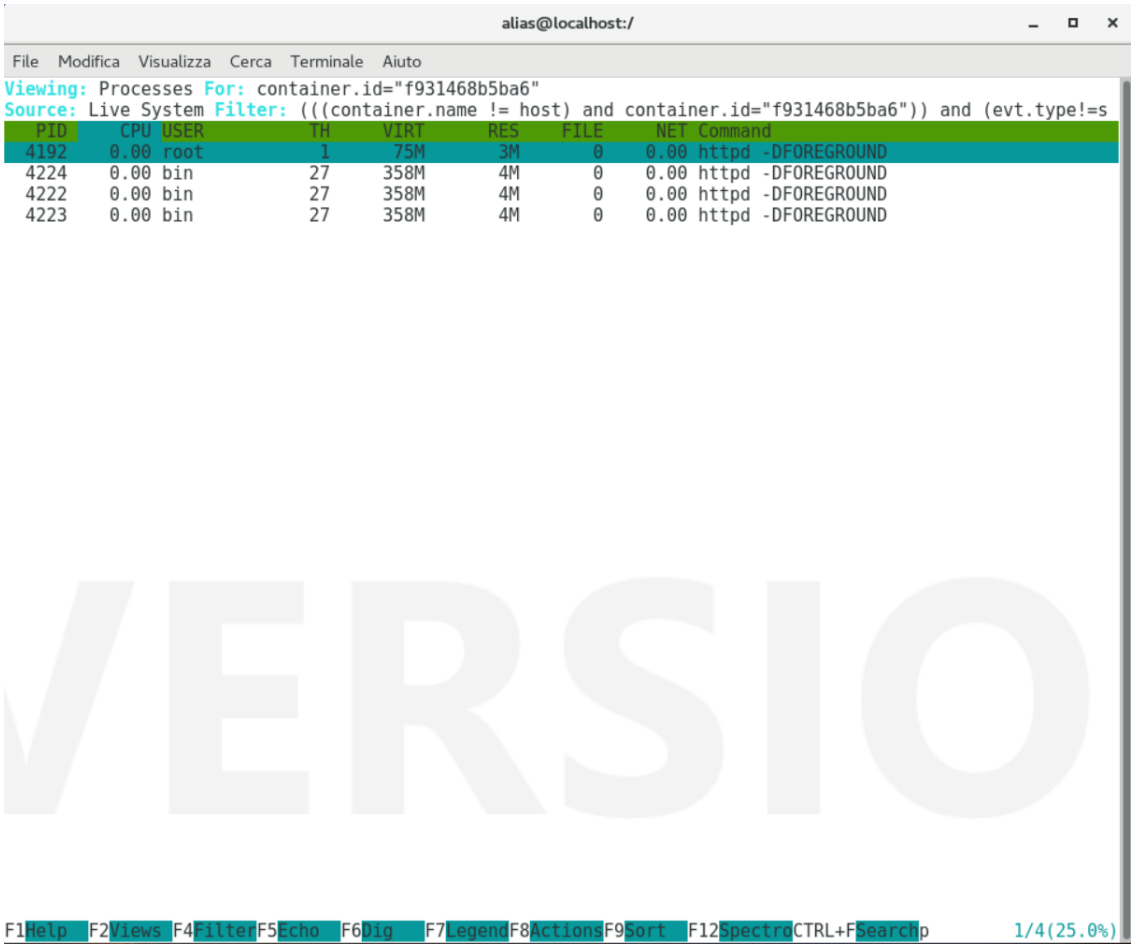

Figura 3.13. Csysdig container list.

fine, è solo necessario eseguire un nuovo riavvio del container e montare nuovamente il volume in questione.

Keep Docker up to date
Tenere aggiornato il container engine è molto importante al fine di risolvere possibili errori nel codice base ed incrementare le performance di esecuzione.

Remove all non-essential services from the host - Network
Idealmente ogni container dovrebbe eseguire solo un servizio. Quindi se il nostro container resta in ascolto su più porte, questo può implicare una violazione di tale principio, a meno che non sia strettamente necessario che tale container resti in ascolto su più di una porta.

Restrict network traffic between container
Come impostazione predefinita, il traffico di rete e abilitato senza alcuna restrizione tra tutti i container dello stesso host. Quindi se i container non hanno bisogno di parlarsi fra loro, sarà buona prassi disattivare tale funzionalità, aggiungendo -icc = false a DOCKER_OPTS, quando dobbiamo avviare il Docker daemon, poiché in caso contrario ogni container ha la possibilità intercettare tutti i pacchetti della rete instaurata tra i container dello stesso host, portando alla divulgazione involontaria e indesiderata delle informazioni di altri container. Per questo motivo risulta molto importante limitare allo stretto necessario la comunicazione tra i container.

Set the logging level
Questo warning ci suggerisce di specificare il livello di logging, quando inizia il daemon docker.

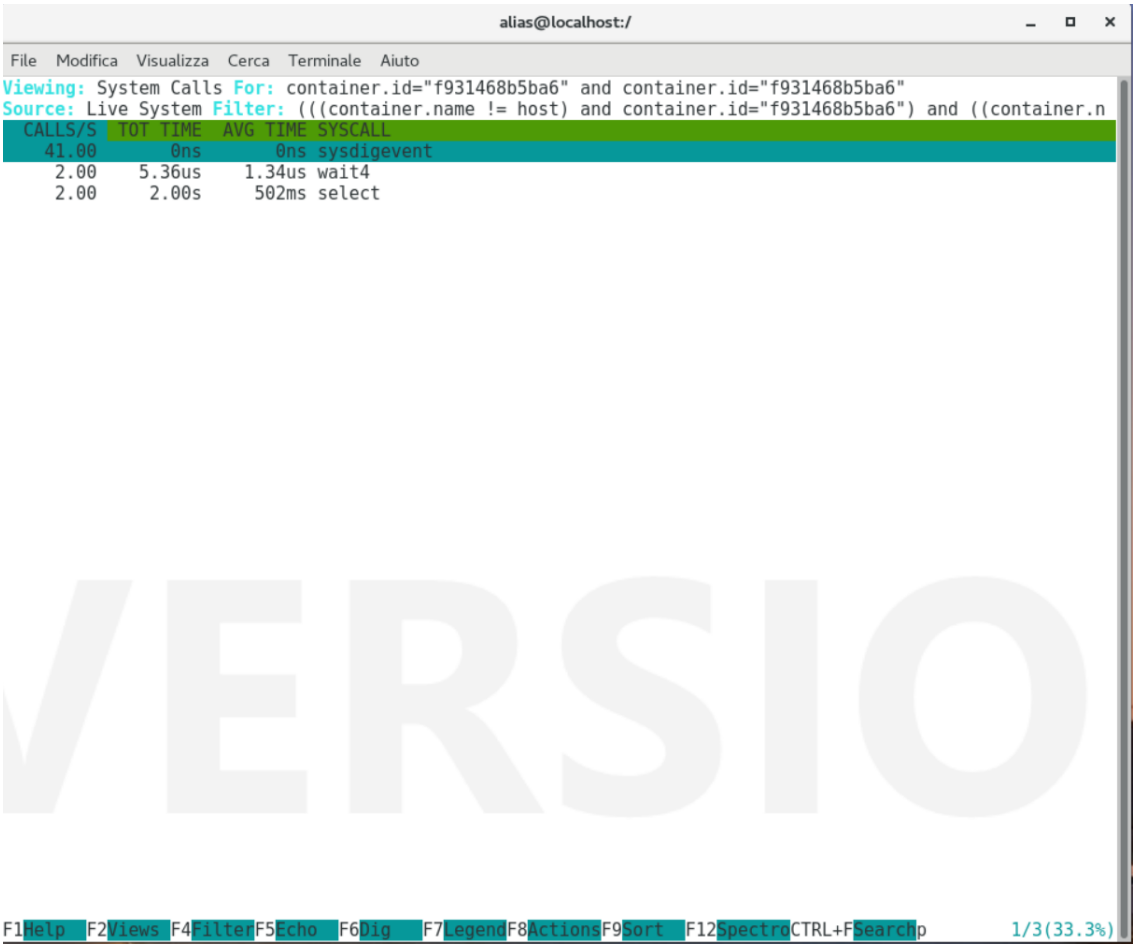

Figura 3.14. Csysdig per container.

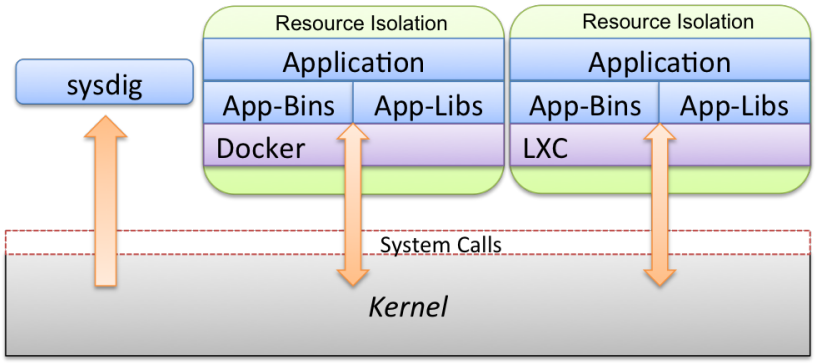

Figura 3.15. Csysdig per container syscall [37].

Di solito possiamo aggiungere “-l log” a DOCKER_OPTS. Per Ubuntu/Debian, controllare /etc/default/docker.

Do not use privileged container
Come detto nei capitoli precedenti, è importante non avviare i container con privilegi a meno che non sia strettamente necessario. E’ quindi importante evitare l’utilizzo del comando “-privileged” quando vogliamo avviare un container.

Do not run ssh within container

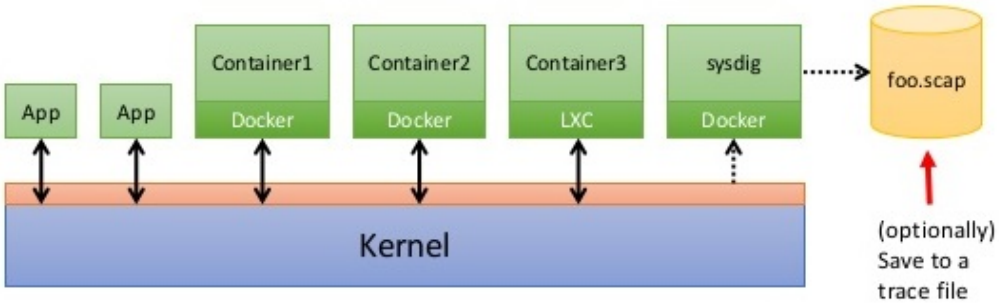

Figura 3.16. Sysdig Architecture internal to a container [38].

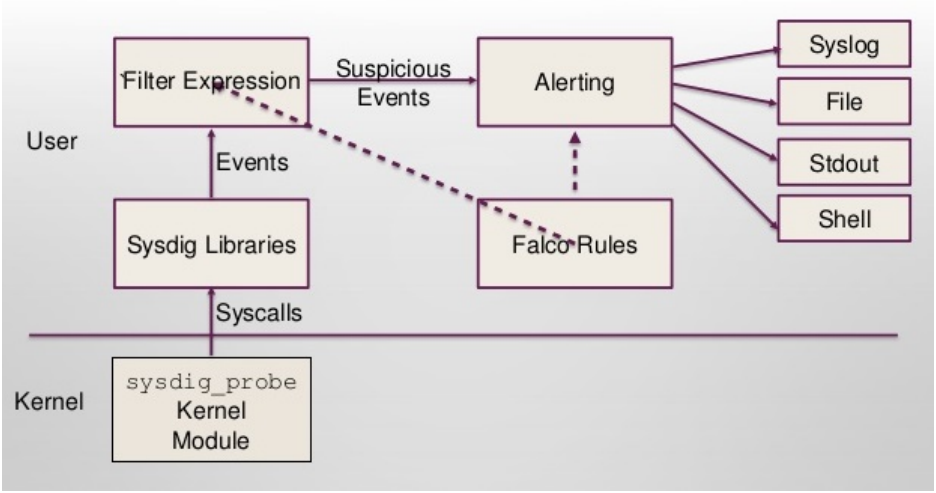

Figura 3.17. Sysdig Falco Architecture [39].

Solitamente si cerca di eseguire un server SSH, perché esso fornisce un modo semplice per "entrare" nel container. La maggior parte di noi lo usa quotidianamente e ha familiarità con le chiavi pubbliche, private, e gli accessi senza password. Tenendo in considerazione quando è stato appena detto, risulta "apparentemente" vantaggioso eseguire un server SSH per permettere la comunicazione tra i container. Ma questo potrebbe non essere completamente vero. Infatti proviamo a fare un esempio pratico, supponendo di voler costruire un' immagine Docker per un server Redis o un webservice Java;

Probabilmente, si desidera eseguire backup, controllare i registri, riavviare il processo, modificare la configurazione, eventualmente eseguire il debug del server con gdb, strace o strumenti simili. Vedremo in seguito come fare queste cose senza SSH.

Risulterà inoltre necessari gestire le proprie chiavi e le password che molto probabilmente verranno salvate in un volume o saranno contenute all'interno di un'immagine. Nel caso in cui queste chiavi o password debbano essere aggiornate, se salvate dentro un'immagine, questa andrà ricostruita ed il container che eseguiva tale immagine andrà eseguito nuovamente. Questo non è un grosso problema, ma di certo non è molto elegante. Una soluzione Migliore potrebbe essere quella di inserire queste credenziali in un volume, per poi gestire il volume stesso. Questa soluzione risulta migliore della precedente, ma presenta alcuni inconvenienti molto gravi. Infatti è necessario assicurarsi che il container non disponga dell'accesso di scrittura al volume, altrimenti, si potrebbero danneggiare le credenziali, impedendo l'accesso stesso al container, o problema ancor peggiore sarebbe se tali credenziali fossero condivise tra più container. Quindi se SSH in questo caso non fosse abilitato tra i container, non ci sarebbe la divulgazione delle credenziali in

Available Container Security Features				
	docker	rkt	LXC	LXD
User Namespaces	Optional from v1.11	Experimental from v1.3	Default from v2.0	Default
Root Capability Dropping	Default from v1.11	Weak default from v1.3	Weak default from v2.0	Default
Procfs and Sysfs Limits	Default from v1.11	Weak default from v1.3	Default from v2.0	Default
Cgroup	Default from v1.11	Weak default from v1.3	Default from v2.0	Default
Seccomp Filtering	Default from v1.11	Optional from v1.3	Weak default from v2.0	Default
Custom Seccomp Filter	Optional	Optional	Optional	Default
Bridge Networking	Default from v1.11	Default from v1.3	Default from v2.0	Default
Hypervisor Isolation	Not possible	Optional from v2.0	Not possible	Default
LSM: AppArmor	Default from v1.11	Not possible	Default from v2.0	Default
LSM: SELinux	Optional from v1.11	Default	Optional from v2.0	Optional
No New Privileges	Optional from v1.11	Not possible	Not possible	Not possible
Container Image Signing	Default from v1.11	Default from v1.3	Default from v2.0	Default

Tabella 3.1. Container Security Features. Tabella ipsirata da [40].

altri container.
Un altro problema è dovuto agli aggiornamenti della protezione.
Infatti, nonostante il server SSH sia abbastanza sicuro, quando si verifica un problema relativo alla sicurezza, sarà necessario aggiornare tutti i container utilizzando SSH.
Ciò implica il dover ricostruire e riavviare tutti i container, e significa anche che si avesse bisogno di utilizzare un servizio relativamente innocuo, sarà comunque necessario mantenere il server SSH sempre aggiornato, poiché la superficie di attacco del container si è ampliata alla superficie di tutti i container che comunicano tra loro tramite SSH. E'inoltre importane sapere che non basta il solo server SSH. Infatti per poter funzionare al meglio, è necessario aggiungere un gestore di processi, come ad esempio Monit o Supervisor.
Questo perché Docker osserva solo un unico processo e quindi un unico container per volta.
Quindi quando un'applicazione si arresta in modo atteso o inatteso, invece di ottenere le informazioni relative all'arresto tramite Docker, queste dovranno essere reperite dal gestore dei processi.
Un altro problema nell'utilizzo di un server SSH si ha quando si devono gestire dei container multi-tenant, dove ogni container è dedicato ad un cliente diverso. E' quindi molto importante che i container non possano comunicare tra loro.

Limit memory usage for container
Come ho spiegato nei capitoli precedenti, è necessario limitare l'utilizzo della memoria in ciascun container con cgroups, in modo tale da evitare attacchi di tipo DoS.

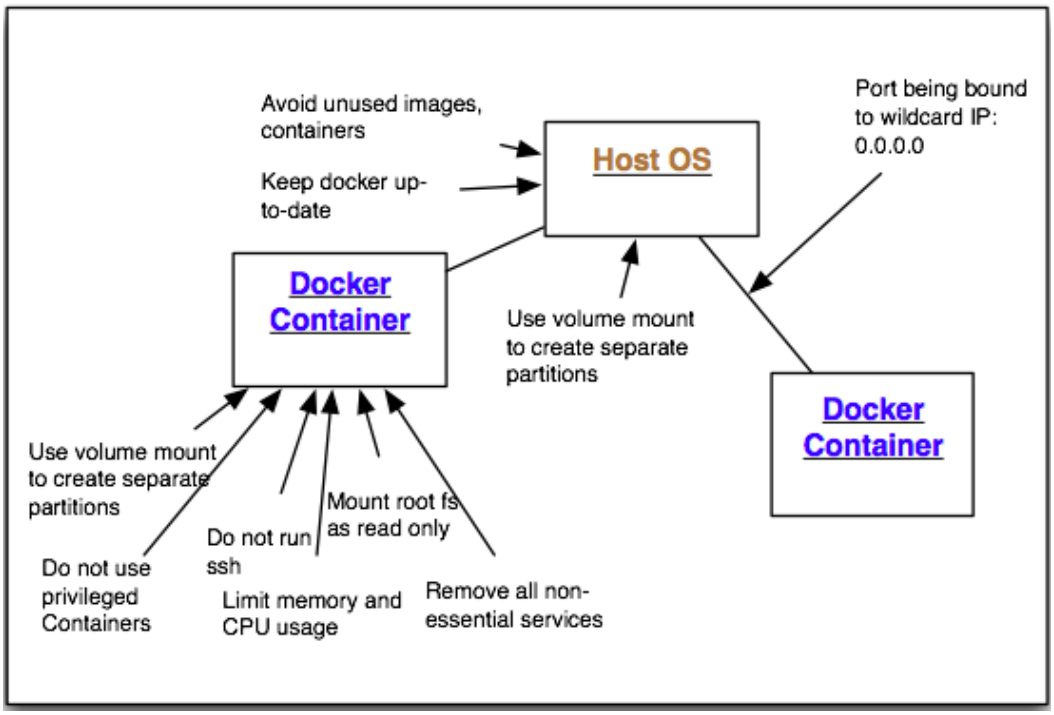

Figura 3.18. Docker Bench Security [41].

Set container CPU priority appropriately
Allo stesso modo è necessario limitare anche le risorse relative alla CPU.

Bind incoming container traffic to a specific host interface

Avoid image sprawl
La rimozione di immagini non necessarie o inutilizzate, può aiutare a risparmiare spazio sul disco.

Container Security Monitoring Tools				
	docker	rkt	LXC	LXD
Sysdig	Optional	Optional	Optional	Optional
Falco	Optional	Optional	Optional	Optional
Docker BenchSecurity Module	Default from v1.11	Not possible	Not possible	Not possible

Tabella 3.2. Container Security Monitoring Tools

Capitolo 4

Analisi pratica della sicurezza nei container relativa a varie implementazioni

In questo capitolo descriverò con l'aiuto di alcuni esempi pratici le soluzioni di sicurezza che ho presentato nel capitolo 3 e comparandole con i vari software di container management basati su kernel Linux, dimostrando in modo pratico ciò che ciascuno di essi può o non può fare per garantire un buon livello di sicurezza nei container.

4.1 CGroups

Come descritto nella sezione 3.1.2, i control groups rappresentano un insieme di processi che sono legati dagli stessi criteri. Questi gruppi possono essere di tipo gerarchico, in cui ogni gruppo eredita dei limiti dal proprio gruppo di appartenenza. Il kernel quindi, si dovrà occupare di fornire l'accesso a più controllori o sottosistemi, attraverso l'interfaccia di cgroups.

Limitare le risorse della memoria fisica in Docker

Di seguito è descritto un esempio implementativo su come limitare le risorse della memoria su ciascun container al fine di poter evitare possibili attacchi di Denial of Services.

```
# Avvio di un container Docker
# abilitato all'utilizzo di non più di 600MB.

docker run -t -d --privileged -h mytest --name my-test \
-m 600M ubuntu:14.04 /bin/bash

docker exec -it my-test bash

# Installazione di un tool per effettuare uno stress test della ram.

apt-get -y update
apt-get -y install stress

#Simulazione utilizzando 500MB di memoria:
in questo caso va tutto a buon fine.

stress --vm-bytes 500m --vm-keep -m 1
```

```
# Simulazione utilizzando 1000MB di memoria:
# in questo caso si verificherà un crash nel container.

stress --vm-bytes 1000m --vm-keep -m 1


# Output:
# stress: info: [83] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
# stress: FAIL: [83] (416) <-- worker 84 got signal 9
# stress: WARN: [83] (418) now reaping child worker processes
# stress: FAIL: [83] (452) failed run completed in 2s
```

Limitare le risorse della memoria di swap in Docker

Un altro utile accorgimento può essere quello di limitare anche la memoria di swapping, ovvero quando si raggiunge il limite della ram, il processo in esecuzione all'interno del container andrà ad utilizzare uno spazio dedicato sul disco. Il dover accedere sul disco però implica maggiori rallentamenti, poichè tale dispositivo a dei tempi di accesso ben più alti della ram, sia nel caso della lettura che della scrittura.

Con l'utilizzo del flag –memory-swap si può fare quanto detto di cui sopra, inoltre, se esso è impostato su un intero positivo, sarà necessario impostare sia il flag –memory-swap che il flag memory–memory. –memory-swap rappresenta la quantità totale di memoria di swapping utilizzabile e –memory ne controlla la quantità utilizzata dalla memoria non swap.

Quindi, se –memory = "300MB" e –memory-swap = "1GB" , il container può utilizzare 300MB di memoria e 700m (1GB - 300MB) di scambio.

Nel caso in cui –memory-swap fosse impostato a 0, tale impostazione verrebbe ignorata considerando un valore di unset.

Se invece il flag –memory-swap venisse impostato allo stesso valore di –memory e –memory fosse impostato su un intero positivo, il container non avrebbe alcun accesso allo swap.

Disattivando –memory-swap ma mantenendo –memory impostato, il container può invece utilizzare il doppio per lo swap. Ad esempio, se –memory = "300MB" e –memory-swap non è impostato, il container potrà utilizzare 300MB di memoria e 600MB di swap.

Se il -memory-swap è esplicitamente impostato su -1, il container può utilizzare un swap illimitato, fino alla quantità disponibile sul sistema host.

E' possibile inoltre, evitare che un container utilizzi lo swapping. Ovvero, se –memory e –memory-swap sono impostati ad uno stesso valore, questo impedirà ai container di utilizzare qualsiasi swap. Questo è dovuto al fatto che –memory-swap è la quantità di memoria fisica combinata a quella di swap che può essere utilizzata, mentre –memory è solo la quantità di memoria fisica utilizzabile. Di seguito elenco alcuni dettagli di swappiness di memoria.

- Un valore di 0 disattiva lo scambio di pagine anonimo.
- Un valore di 100 imposta tutte le pagine anonime come swap.

Come impostazione predefinita, se non si imposta –memory swappiness, il valore viene ereditato dalla macchina host.

Limiti di memoria del kernel in Docker

I limiti di memoria del kernel sono espressi in termini di memoria complessiva assegnata ad uno specifico container. Di seguito descriverò i diversi scenari di configurazione che si possono andare ad ottenere.

- Memoria illimitata, memoria del kernel illimitata: questo è il comportamento che viene impostato di default.
- Memoria illimitata, memoria limitata del kernel: questo è appropriato quando la quantità di memoria necessaria da tutti i gruppi è maggiore della quantità di memoria effettivamente presente sulla macchina host. È possibile configurare la memoria del kernel per non superare mai ciò che è disponibile sulla macchina host e i container che necessitano di più memoria devono attenderlo.
- Memoria limitata, memoria del kernel umido: La memoria complessiva è limitata, ma la memoria del kernel non è.
- Memoria limitata, memoria limitata del kernel: la limitazione della memoria dell'utente e del kernel può essere utile per il debug di problemi legati alla memoria. Se un container utilizza una quantità imprevista di entrambi i tipi di memoria, esaurirà la memoria senza compromettere altri container o la macchina host. All'interno di questa impostazione, se il limite di memoria del kernel è inferiore al limite della memoria utente, l'esaurimento della memoria del kernel provocherà un errore OOM. Se il limite di memoria del kernel è superiore al limite della memoria utente, il limite del kernel non provocherà il container a verificare un OOM.

Quando si attiva qualsiasi limite di memoria del kernel, la macchina host visualizza le statistiche "high water mark" su base di processo, in modo da poter monitorare quali processi (in questo caso, quelli in esecuzione all'interno dei container) stanno utilizzando la memoria in eccesso. Questo può essere visto per processo visualizzando lo stato /proc/PID/ status sulla macchina host.

Limitare le risorse della CPU in Docker

Di default, l'accesso di ciascun container ai cicli CPU della macchina host è illimitato. È possibile infatti, impostare vari vincoli per limitare l'accesso di un determinato container ai cicli di CPU della macchina host. Nella maggior parte dei casi si andrà ad utilizzare e configurare lo scheduler CFS di default. Il CFS è quindi lo scheduler CPU del kernel Linux per i processi Linux normali. Alcuni flags di runtime consentono di configurare la quantità di accesso alle risorse della CPU di un dato container. Quando si utilizzano queste impostazioni, Docker modifica le impostazioni per il gruppo cgroup del container sulla macchina host.

Se ad esempio volessimo limitare le risorse di un container Docker a quelle relative al primo core della CPU, potremmo utilizzare il comando –cpuset-cpus = 0.

È inoltre possibile utilizzare l'argomento –group-parent e impostare manualmente vincoli di risorse ad un livello di granularità maggiore.

Nell'esempio sottostante ho utilizzato 2 differenti tipi di container Docker, ('Container1' e 'Container2'), allo scopo di avviare un container di base 'busybox' con md5sum/dev/urandom e simulare un processo che richieda un elevato quantitativo di risorse CPU. Infatti per impostazione di default, tale processo arriverebbe a consumare tutte le risorse CPU disponibili. Quindi applicherò due criteri Cgroup per gestirne le risorse. Innanzitutto utilizzero 'cpuset.cpus' per bloccare i container allo stesso core CPU (core 0). Successivamente utilizzerò 'cpu.shares' per assegnare una relativa quota di CPU, dando al Container1 un valore di 20 ed al Container2 un valore di 80. Questo implica che il 20% della CPU sarà assegnato al Container1, mentre l'80% della CPU sarà assegnato al Container2, come visualizzato nella figura **??**

```
$ docker run -d --name='Container1' --cpuset-cpus=0 --cpu-shares=20
busybox md5sum /dev/urandom
$ docker run -d --name='Container2' --cpuset-cpus=0 --cpu-shares=80
busybox md5sum /dev/urandom
```

Richiamando invece i container, senza l'utilizzo dei precedenti comandi, si ottiene come risultato che entrambi i core della cpu vengono utilizzati al 100% come visualizzato nella figura **??**

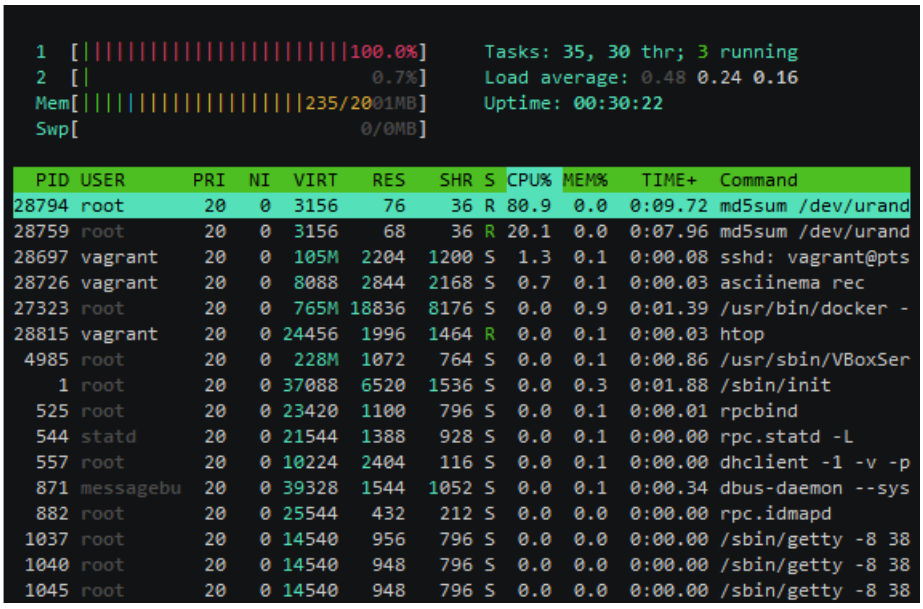

Figura 4.1. Htop: Cgroup CPU limitations. Si può notare dalla barra rossa presente nella parte in alto a sinistra, che solo la cpu 1 viene utilizzata al 100% a differenza della cpu 2 che non viene quasi utilizzata

```
$ docker run -d --name='Container1' busybox md5sum /dev/urandom
$ docker run -d --name='Container2' busybox md5sum /dev/urandom
```

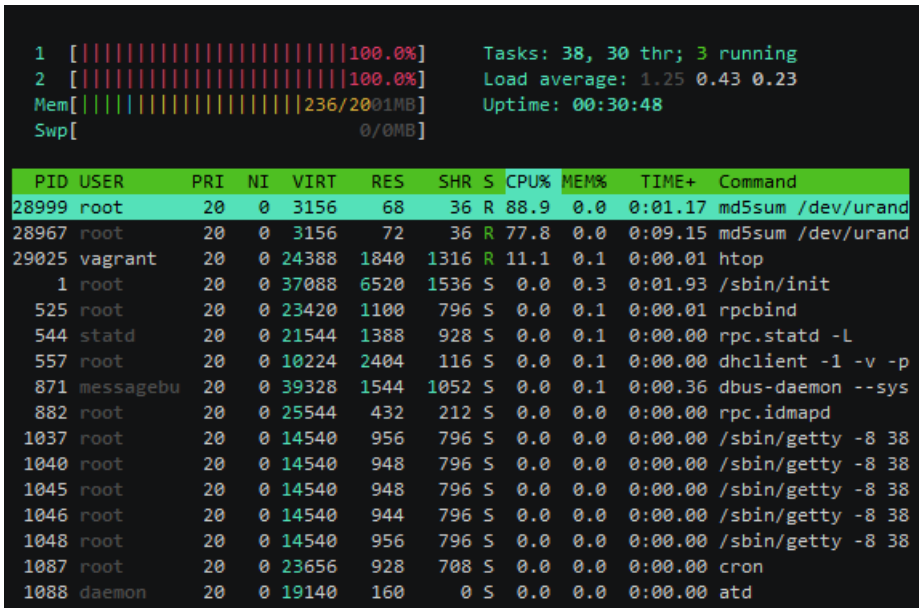

Figura 4.2. Htop: Cgroup NO CPU limitations. In questo caso, si può notare da entrambe le barre rosse presenti nella parte in alto a sinistra, che tutte e 2 cpu vengono utilizzate al 100%

Limitare le risorse della CPU in LXC ed LXD

Attualmente la gestione delle risorse nei container Linux (LXC) è affrontata da LXD, progetto costruito in cima ad LXC. LXD offre quindi, una serie di opzioni per controllare le risorse dei Container Linux e per impostare, quando questo risultasse necessario anche i limiti di tali risorse

per ciascun container. Di seguito descriverò i meccanismi principali utili ad impostare i vincoli di CPU e memoria. La gestione della CPU viene eseguita in 1 dei seguenti 4 modi, a seconda del carico di lavoro previsto e del regime di gestione dell'host CPU.

- Numero di CPU - E' possibile impostare il numero di cores della CPU che LXC può utilizzare con uno specifico container, permettendo inoltre di distribuire automaticamente il tempo di CPU tra gli host, nei casi in cui vi è un contest per il tempo della CPU.
- Core specifici - E' possibile definire specifici cores fisici per il container allo scopo di utilizzare e distribuire il tempo di CPU disponibile tra i vari container, per i casi in cui più container concorrono per utilizzare gli stessi cores.
- Capped sharing - Permette di definire una specifica percentuale di tempo di CPU per uno o più container. Quando l'host non è sotto carico, allora un container può utilizzare qualsiasi CPU disponibile, ma tuttavia, quando c'è il contest per la CPU, allora il container sarà limitato alla quantità specificata.
- Condivisione a tempo limitato - Questa modalità è simile a quella precedente in quanto è possibile vedere tutte le CPU, ma questa volta, è possibile utilizzare solo il tempo di CPU più lungo impostato nel limite, indipendentemente dal fatto che il sistema sia inattivo. Su un sistema senza alcun sovraccarico questo consente di tagliare la CPU in modo accurato, garantendo ai container delle prestazioni costanti.

L'impostazione dei limiti avviene con il comando lxc. Ci sono poi due opzioni; limite.cpu utilizzabile per i primi 2 punti descritti di sopra e limit.cpu.allowance per gli ultimi 2 punti.

```
lxc config set [CONTAINER] limits.cpu [VALUE]

lxc config set [CONTAINER] limits.cpu.allowance [VALUE]
```

Di seguito descriverò un esempio di come gestire le risorse CPU nei container Linux. Imposto un container nginx-proxy per utilizzare una qualunque delle 2 CPU presenti nell'host.

```
lxc config set nginx-proxy limiti.cpu 2
```

Ora imposto il container nginx-proxy per utilizzare il 20

```
lxc config set nginx-proxy limiti.cpu.allowance 20%
```

Imposto nginx-proxy del container per utilizzare non più del 50

```
lxc config set nginx-proxy limiti.cpu.allowance 100ms / 200ms
```

L'ultima opzione utile alla limitazione della CPU è la priorità del tempo di CPU. Questa opzione viene utilizzata quando le risorse CPU dell' host sono andate in overload e i container sono in contest per il tempo di CPU. I valori disponibili sono compresi tra 0 e 10 e i numeri più bassi indicano una priorità inferiore, invece un numero maggiore significa che si otterrà del tempo di CPU prima dei numeri inferiori. Il comando sotto imposta il nginx-proxy del container per avere una priorità CPU di 5.

```
lxc config set nginx-proxy limits.cpu.priority 5
```

Il comando di seguito, imposterà il php-backend del container per avere una priorità della CPU di 2 e pertanto avrebbe meno tempo di CPU rispetto a quello del container nginx-proxy quando la CPU è in fase di contest.

```
lxc config set php-backend limits.cpu.priority 5
```

Limitare le risorse della memoria in LXC ed LXD

Riguardo all'utilizzo della memoria, si possono definire dei limiti della stessa, basati su percentuali. Inoltre è possibile scegliere di attivare o disattivare lo swap per ogni container e, nel caso in cui fosse abilitato, è possibile impostare una priorità in modo da poter scegliere quale container sarà abilitato per primo ad eseguire lo swap-out sul disco. I limiti di memoria sono definiti "hard" di default. Ovvero, quando si esaurisce la memoria, il kernel-out-of-memory-killer, inizierà a terminare i processi in esecuzione, sinon a rientrare nei limiti della memoria. In alternativa è anche possibile impostare la politica di applicazione in "soft" , nel qual caso sarà consentito utilizzare la memoria desiderata finché non verrà completamente esaurita. Non appena qualcun altro avrà necessità di memoria, al container configurato con l'impostazione "soft" non sarà più possibile allocarne, fino a che non rientrerà nei limiti di memoria o fino a quando l'host non avrà altra memoria disponibile. Per poter applicare un corretto limite di memoria utilizzerò i comandi descritti di seguito.

Con questo primo comando applicherò un limite di 256MB di memoria al container my-container.

```
lxc config set my-container limits.memory 256MB
```

Con questo comando disattiverò lo swap per un dato container. Tale impostazione è abilita di default.

```
lxc config set my-container limits.memory.swap false
```

Con quest altro comando, informerò il kernel il quale dovrà effettuare lo swap di questo container prima di tutti.

```
lxc config set my-container limits.memory.swap.priority 0
```

Con quest' ultimo comando, disabiliterò l'hrd memory limit enforcement.

```
lxc config set my-container limits.memory.enforce soft
```

Limitare le risorse di CPU e memoria in Rkt

Come per le precedenti tecnologie di gestione dei container, anche Rkt fornisce la possibilità di limitare le risorse a disposizione delle immagini di applicazioni avviate all'inerno dei container. Nelle immagini sono già presenti degli isolatori che definiscono le risorse utilizzabili per l'applicazione che dovrà andare in esecuzione all'interno di un container. Nella gran parte dei casi, gli isolatori presenti all'interno delle immagini, possono essere sovrascritti da Rkt. Nell'esempio seguente, descriverò quale comando utilizzare per definire le risorse di CPU e memoria per un dato container.

```
# rkt run coreos.com/etcd:v2.0.0 --cpu=750m --memory=128M
```

Nell'esempio di cui sopra, l'isolatore CPU è definito a 750 millisecondi e l'isolatore di memoria limita l'utilizzo della stessa a 128MB.

4.2 Seccomp

Come descritto nella sezione 3.1.5, Seccomp è una funzionalità del kernel che consente di filtrare le chiamate di sistema provenienti dal container verso il kernel. La combinazione di chiamate limitate e chiamate consentite è disposta in profili, ed è possibilie passare questi diversi profili a diversi container. Seccomp fornisce inoltre un controllo delle funzionalità ad una granularità più alta, dando ad un attaccante un numero limitato di syscall che possono essere chiamate dall' interno di un container.

Seccomp in Docker

L'analisi pratica è stata fatta utilizzando Ubuntu 16.04 e Docker 17.04.0-ce. In questa versione di Docker le componenti di sicurezza installate risultano quelle di seguito.

```
$ docker info | grep seccomp
Security Options: apparmor seccomp
```

Docker utilizza seccomp in modalità filtro e dispone di un proprio DSL basato su JSON che consente di definire e compilare altri profili diversi da quello di default. Quando si esegue un container viene lanciato il profilo di default di seccomp, eccetto nel caso in cui non si utilizzi il flag -security-opt, in aggiunta al comando run docker per avviare un profilo personalizzato.
Con il comando descritto nell' esempio successivo, inizializzo un container interattivo basato sull'immagine Alpine e avviare un processo di shell, applicando inoltre anche il profilo seccomp descritto da ¡profile¿ .json.

```
$ docker container run -it --rm --security-opt
seccomp=<profile>.json alpine sh
```

Il comando di cui sopra invia il file JSON dal client al daemon dove viene compilato in un programma BPF, utilizzando un involucro Go intorno a libseccomp.
I profili Docker seccomp operano utilizzando un approccio whitelist che specifica le syscall consentite. Sono quindi consentiti solo i sistemi di chiamata basati su whitelist.
Poichè Docker supporta altre tecnologie di sicurezza come ad esempio Apparmor, è possibile che esse interferiscano con il test dei profili seccomp. Per questo motivo, il modo migliore per verificare l'effetto dei profili seccomp è quello di aggiungere tutte le funzionalità e disattivare in questo caso Apparmor. Questo è utile per darci la sicurezza che il comportamento che descriverò nei passaggi successivi è dovuto esclusivamente alle modifiche seccomp.
I seguenti flag di esecuzione di Docker aggiungono tutte le funzionalità e disattivano Apparmor.

```
--cap-add ALL --security-opt apparmor = non definito.
```

In questo passaggio si utilizza il profilo deny.json seccomp. Questo profilo ha una whitelist syscall vuota, il che significa che tutte le syscall saranno bloccate. Inoltre si aggiungeranno tutte le funzionalità e si disattiverà Apparmor in modo da sapere che solo il profilo seccomp sta impedendo le syscall. Con il comando run docker verrà avviato un nuovo container con tutte le capabilities abilitate, Apparmor disabilitato tramite il flag unconfined e il profilo seccomp-profiles / deny.json seccomp applicato.

```
$ docker container run --rm -it --cap-add ALL
 --security-opt apparmor=unconfined
 --security-opt seccomp=seccomp-profiles/deny.json alpine sh

docker: Error response from daemon: exit status 1:
"cannot␣start␣a␣container␣that␣has␣run␣and␣stopped\n".
```

In questo scenario, Docker non ha abbastanza syscall per poter avviare il container ed è per questo motivo che viene generato tale errore. Di seguito ho effettuato l'ispezionare del contenuto del profilo seccomp-profiles / deny.json.

```
$ cat seccomp-profiles / deny.json
    {
        "defaultAction": "SCMP_ACT_ERRNO",
        "architetture": [
               "SCMP_ARCH_X86_64",
               "SCMP_ARCH_X86",
               "SCMP_ARCH_X32"
        ],
        "syscalls": [
        ]
    }
```

Da questo profilo è quindi possibile notare che non ci sono syscall nella whitelist. Questo significa che non è consentita alcuna syscall da container avviati con tale profilo. A meno che non si specifichi un profilo diverso, Docker applicherà il profilo di default seccomp a tutti i nuovi container. In questo passaggio vedrò come forzare un nuovo container all'esecuzione senza alcun profilo seccomp. Innanzitutto avvierò un nuovo container con la sigla –security-opt seccomp = unconfined, in modo che non venga applicato alcun profilo seccomp.

```
$ run docker container eseguire --rm -it
--security-opt seccomp = debian non definito: jessie sh
```

Dal terminale del container eseguirò il comando whoami per confermare che il container funziona correttamente e che le syscall ritornano al Docker host.

```
/ # whoami
    root
```

Per dimostrare che non si è in esecuzione con il profilo predefinito seccomp, di seguito ho eseguito il comando unshare, per creare un nuovo namespace.

```
/ # unshare --map-root-utente -user
/ # whoami
    root
```

Tentando invece di eseguire il comando di unshare da un container che ha seccomp abilitato in modalità di default, esso non riuscirà e restituirà un messaggio di errore il quale indica che l'operazione non è consentita.

Eseguendo il comando successivo, si potrà visualizzare un elenco delle system call utilizzate dal programma whoami. E' però necessario che l'host nella quale è in esecuzione Docker abbia il pacchetto Strace installato.

```
$ strace -c -f -S name whoami 2>&1 1>
/dev/null | tail -n +3 | head -n -2 | awk '{print $(NF)}'
access
arch_prctl
brk
close
connect
execve
<SNIP>
socket
write
```

Oppure in alternativa si può utilizzare il comando successivo.

```
$ strace whoami
execve("/usr/bin/whoami", ["whoami", "-qq"], [/* 21 vars */]) = 0
brk(0) = 0x1980000
<SNIP>
```

Ovviamente Strace può essere utilizzato per qualunque altro programma, infatti whoami è solo un esempio.

Con Seccomp è possibile inoltre, rimuovere in modo selettivo i vari tipi di syscall. Nell' esempio seguente, ho modificato il profilo default.json e l'ho rinominato in default-no-chmod.json, in modo da ottimizzare le syscall disponibili per i container. Infatti in questo nuovo profilo, come si può intuire dal nome stesso, rimuoverò le syscall chmod(), fchmod() e chmodat() dalla whitelist. Successivamente avvierò un nuovo container con tale profilo e tenterò l'esecuzione del comando chmod 777 -v.

```
$ run dock container eseguire --rm -it
--security-opt seccomp = default-no-chmod.json alpine sh

/ # chmod 777 / -v
    chmod: /: Il funzionamento non è consentito
```

Il comando non riesce perché chmod 777 -v utilizza alcune delle syscall chmod(), fchmod() e chmodat() che sono state rimossi dalla whitelist del profilo default-no-chmod.json.

Nel passo successivo invece, avvierò un nuovo container con il profilo default.json e manderò in esecuzione lo stesso comando: chmod 777 -v.

```
$ run dock container execute
--rm -it --security-opt seccomp = default.json alpine sh
/ # chmod 777 / -v
```

La modalità '/' è stata quindi modificata a 0777 (rwxrwxrwx)

Questa volta il comando riesce perché il profilo default.json ha le syscall chmod(), fchmod() e chmodat() incluse nella whitelist.

Ora proverò a controllare entrambi i profili per verificare la presenza delle syscall chmod(), fchmod() e chmodat(), eseguendo i comandi successivi, dalla riga di comando del Docker Host e non dall'interno del container creato nel passaggio precedente.

```
$ cat ./seccomp-profiles/default.json | grep chmod
"name": "chmod",
"name": "fchmod",
"name": "fchmodat",
$ cat ./deccomp-profiles/default-no-chmod.json | grep chmod
```

L'output di cui sopra mostra che il profilo default-no-chmod.json non contiene nessuna syscall correlata a chmod all'interno della whitelist, a differenza del profilo di default che le contiene tutte e tre.

Di seguito invece, concentrerò la mia descrizione sul come scrivere un profilo seccomp da zero per Docker.

Il seguente è il layout di un profilo Docker seccomp.

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "name": "accept",
            "action": "SCMP_ACT_ALLOW",
            "args": []
        },
        {
            "name": "accept4",
            "action": "SCMP_ACT_ALLOW",
            "args": []
        },
        ...
    ]
}
```

Action	Description
SCMP_ACT_KILL	Kill with a exit status of `0x80 + 31 (SIGSYS) = 159`
SCMP_ACT_TRAP	Send a `SIGSYS` signal without executing the system call
SCMP_ACT_ERRNO	Set `errno` without executing the system call
SCMP_ACT_TRACE	Invoke a ptracer to make a decision or set `errno` to `-ENOSYS`
SCMP_ACT_ALLOW	Allow

Figura 4.3. La tabella di sopra, elenca le possibili azioni in ordine di priorità. Le azioni più alte eliminano le azioni ai livelli inferiori.

Le azioni più importanti per gli utenti Docker sono dettate da SCMP_ACT_ERRNO e SCMP_ACT_ALLOW.

I profili possono inoltre contenere filtri più granulari in base al valore degli argomenti alla chiamata di sistema, come descritto di seguito.

```
{
...
"syscalls": [
     {
          "name": "accept",
          "action": "SCMP_ACT_ALLOW",
          "args": [
              {
                  "index": 0,
                  "op": "SCMP_CMP_MASKED_EQ",
                  "value": 2080505856,
                  "valueTwo": 0
              }
          ]
       }
   ]
}
```

index è l'indice dell'argomento di chiamata di sistema.
op è l'operazione da eseguire sull'argomento. Può essere uno di:

SCMP_CMP_NE - non uguale
SCMP_CMP_LT - meno di
SCMP_CMP_LE - meno o uguale a
SCMP_CMP_EQ - uguale a
SCMP_CMP_GE - maggiore di
SCMP_CMP_GT - maggiore o uguale a
SCMP_CMP_MASKED_EQ - true if (valore AND arg == valoreTwo)

value è un parametro per l'operazione.
valueTwo invece, viene utilizzato solo per SCMP_CMP_MASKED_EQ.

Utilizzando Strace, come detto in precedenza, è possibile ottenere un elenco di tutte le chiamate di sistema effettuate da un programma.

Seccomp in LXC

In LXC la procedura di configurazione è differente. Infatti è possibile avviare un container con una ridotta serie di chiamate di sistema, che sono disponibili caricando un profilo seccomp all'avvio. Il file di configurazione seccomp deve sempre iniziare con un numero di versione nella prima riga, un tipo di criterio nella seconda riga, seguito dalla configurazione.

Dalla versione 2.0 di LXC c'è il pieno supporto di seccomp versione 1 e 2. Nella versione 1, la politica è un semplice whitelist. La seconda riga deve dunque contenere "whitelist" , e nel resto del file si aggiunge in ogni riga, un numero che indica la syscall che si vuole sia abilitata all'interno del container Linux. Ogni numero di syscall non presente nella "whitelist" risulterà in blacklist, e quindi tali syscall non saranno permesse all'interno del container.

Nella versione 2 invece, la politica può essere o una whitelist o una blacklist, inoltre supporta le azioni predefinite per-rule e per-policy e supporta la per-architecture syscall resolution da nomi testuali.

Di seguito mostrerò un semplice esempio di criterio di blacklist in cui sono consentite tutte le chiamate di sistema tranne che per mknod, che non farà nulla e restituirà 0 (successo).

```
2
blacklist
mknod errno 0
```

Tale criterio è descritto in un file di profilo per seccomp, che sarà utilizzato prima di avviare un container utilizzando il comando lxc.seccomp.profile.

Seccomp in LXD

In LXD invece, tutti i container sono limitati da una policy di default di seccomp. Questa policy impedisce alcune azioni pericolose come gli umount forzati, il caricamento e lo scaricamento del modulo del kernel, kexec e la chiamata di sistema open_by_handle_at. La configurazione seccomp non può essere modificata, ma può essere tuttavia utilizzata una policy completamente differente o nessuna policy, utilizzando raw.lxc. Questo può esser necessario quando ad esempio si vuole comunicare con il driver lxc sottostante. Il tutto può esser fatto specificando le voci di configurazione LXC nella chiave di configurazione LXD 'raw.lxc'.

Seccomp in Rkt

Rkt, a differenza degli altri container management system, viene fornito con una serie di gruppi di filtraggio predefiniti che possono essere utilizzati per creare rapidamente ambienti sandboxed per applicazioni che andranno in esecuzione all'interno di container. Ogni set è semplicemente un riferimento a un gruppo di syscall che copre una singola area funzionale o un sottosistema del kernel. Possono essere ulteriormente combinati per creare filtri più complessi, sia per la lista nera che per il whitelisting di chiamate di sistema specifiche. Per distinguere questi gruppi predefiniti da nomi di syscall reali, le etichette wildcard sono precedute da un prefisso che corrisponde al simbolo @ oltre ad essere namedpaced.

l'*Application Container Specification*(Appc) definisce due gruppi:

```
@ appc.io / all
```

Rappresenta l'insieme di tutte le syscall disponibili.

```
@ appc.io / empty
```

Rappresenta l'insieme vuoto. Rkt a sua volta fornisce due gruppi predefiniti per un utilizzo generico:

```
@ rkt / default-blacklist
```

Questo rappresenta un filtro di ampia portata rispetto a quello che può essere utilizzato per blacklisting generico.

```
@ rkt / default-whitelist
```

Quest'altro invece, rappresenta un filtro di ampia portata rispetto a quello utilizzabile per il whitelisting generico. Per motivi di compatibilità, vengono forniti anche due gruppi di profili Docker predefiniti:

```
@docker/default-blacklist
@docker/default-whitelist
```

Quando si utilizzano immagini stage1 con systemd¿ = v231, sono disponibili anche alcuni gruppi predefiniti:

```
@ systemd / clock
```

Disponibile per syscall che manipolano l'orologio di sistema.

```
@ systemd / whitelist
```

Disponibile di default per un insieme generico di syscall tipicamente autorizzate.

```
@ systemd / mount
```

Utile per il montaggio e lo smontaggio del filesystem.

```
@ systemd / network-io
```

Utilizzato per operazioni di I / O socket.

```
@ systemd /
```

Utilizzato per syscall obsolete,inusuali,o non implementate.

```
@ systemd / privileged
```

Per i sistemi di comunicazione che necessitano di syscall super-user.

```
@ systemd / process
```

Per syscall che agiscono sul controllo di processo, sull'esecuzione e la denominazione.

```
@ systemd / raw-io
```

Per l'accesso raw alle porte di I/O.

Quando non è specificato alcun filtro seccomp, di default, vi è il whitelisting di tutte le syscall tipicamente necessarie alle applicazioni per eseguire le operazioni più comuni. Questo è lo stesso insieme di syscall che viene definito da @ rkt / default-whitelist.

Tipicamente , l'impostazione predefinita è adattata per impedire alle applicazioni di eseguire una grande varietà di azioni privilegiate, senza peraltro influenzare il loro comportamento normale. Invece le operazioni che in genere non sono necessarie nei container e che possono influenzare lo stato di accoglienza, ad es. invocando umount(2), vengono negate.

Tuttavia, questo set di default è per lo più inteso come precauzione di sicurezza contro le applicazioni erratiche e inadeguate e non è sufficiente a bloccare attacchi sviluppati su misura. Per tale motivo è sempre meglio utilizzare il filtro seccomp utilizzando uno degli isolatori personalizzabili disponibili in rkt.

Quando si esegue un container Linux, rkt fornisce due isolatori reciprocamente esclusivi per definire un filtro seccomp per un'applicazione.

```
os / linux / seccomp-retain-set
os / linux / seccomp-remove-set
```

Questi isolatori coprono diversi casi d'uso e impiegano tecniche diverse per raggiungere lo stesso obiettivo, ovvero quello di limitare i sistemi di comunicazione disponibili, poichè in quanto tali, non possono essere utilizzati contemporaneamente, e l'uso consigliato varia caso per caso.

Gli isolatori Seccomp lavorano definendo una serie di syscall che possono essere bloccate "remove-set" o consentite "retain-set" . Una volta che un'applicazione tenta di richiamare un sistema syscall bloccato, il kernel negherà questa operazione e verrà notificato un errore dell'applicazione.

Di default infatti,l'invocazione di syscall bloccate comporterà l'immediata conclusione dell'applicazione con un segnale di SIGSYS. Questo comportamento può essere modificato restituendo all'applicazione un codice di errore specifico "errno" invece di terminarla.

Per entrambi gli isolatori, questo può essere personalizzato specificando un parametro aggiuntivo errno con il nome errno simbolico desiderato.

Quindi os/linux/seccomp-keep-set consente di utilizzare un approccio additivo per creare un filtro seccomp. Infatti le applicazioni non sono in grado di utilizzare qualsiasi syscall, ad eccezione di quelle elencate all'interno di questo isolatore.

Questo approccio whitelisting è utile per chiudere completamente gli ambienti e quando i requisiti applicativi "in termini di syscall" sono ben definiti in anticipo.

Ad esempio, il "retain-set" per un'applicazione di rete tipica, includerà le voci per le operazioni POSIX generiche (disponibili in @ systemd/default-whitelist), operazioni socket (systemd/network-io) e la reazione agli eventi di I/O come systemd/io-evento).

Invece os/linux/seccomp-remove-set, setta le syscall in modo sottrattivo. Infatti, partendo da tutti i sistemi di accesso disponibili, è possibile vietare singole voci per evitare azioni specifiche.

Questo approccio di tipo blacklisting è utile per limitare in qualche modo applicazioni che dispongono di ampie richieste in termini di syscall, al fine di negare l'accesso a sistemi di chiamata chiaramente inutilizzati ma potenzialmente sfruttabili.

Ad esempio, un'applicazione che dovrà eseguire più operazioni, ma che allo stesso tempo non può toccare i punti di montaggio, potrebbe avere @ systemd/mount, con specificato "remove-set" .

Di seguito descriverò alcuni esempi pratici.

L'obiettivo di questi esempi è quello di mostrare come creare immagini ACI con acbuild, in cui alcune delle syscall vengono bloccate o consentite in modo esplicito. Per semplicità, il punto di partenza sarà l'uso di un'immagine alpine Linux che viene fornita con comandi di ping e umount (da busybox). Questi comandi rispettivamente richiedono syscall(2) e umount(2) per eseguire operazioni privilegiate. Per bloccare il loro utilizzo, un filtro syscall può essere installato tramite os/linux/seccomp-remove-set o os/linux/seccomp-keep-set. Di seguito mostrerò entrambi gli approcci.

Questo primo esempio mostra come bloccare l'operazione di socket (ad esempio con ping), rimuovendo socket() dall'insieme delle syscall consentite.

In primo luogo, un'immagine locale è costruita con un isolatore "remove-set" esplicito. Questo set contiene le syscall che devono essere vietate per bloccare l'impostazione del socket.

```
$ acbuild begin
$ acbuild set-name localhost/seccomp-remove-set-example
$ acbuild dependency add quay.io/coreos/alpine-sh
$ acbuild set-exec -- /bin/sh
$ echo '{ "set": ["@rkt/default-blacklist", "socket"] }' |
acbuild isolator add "os/linux/seccomp-remove-set" -
$ acbuild write seccomp-remove-set-example.aci
$ acbuild end
```

Una volta che questa immagine è stata costruita correttamente, può essere eseguita per verificare che l'utilizzo di ping sia bloccato dal filtro seccomp. Allo stesso tempo, la blacklist predefinita blocca anche altre syscall pericolose come umount(2):

```
$ sudo rkt run --interactive --insecure-options=image
seccomp-remove-set-example.aci
image: using image from file stage1-coreos.aci
image: using image from file seccomp-remove-set-example.aci
image: using image from local store for image name quay.io/
coreos/alpine-sh

# whoami
root

# ping -c1 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
Bad system call

# umount /proc/bus/
Bad system call
```

Ciò significa che le socket(2) e umount(2) sono state effettivamente disattivate all'interno del container.

A differenza dell'esempio precedente, di seguito mostrerò come consentire solo alcune operazioni (ad esempio, la comunicazione di rete via ping), consentendo di abilitare tutti i sistemi di chiamata richiesti. Ciò significa che i sistemi di chiamata esterni a questo set saranno bloccati.

In primo luogo, un'immagine locale è costruita con un esplicito isolatore di "keep-set" . Questo set contiene quindi, la "whitelist" predefinita di jctc di rkt (che già fornisce tutte le voci relative a socket) e alcune syscall personalizzate, come ad esempio, umount (2), che non è tipicamente consentita:

```
$ acbuild begin
$ acbuild set-name localhost/seccomp-retain-set-example
$ acbuild dependency add quay.io/coreos/alpine-sh
$ acbuild set-exec -- /bin/sh
$ echo '{ "set": ["@rkt/default-whitelist", "umount", "umount2"] }'
 | acbuild isolator add "os/linux/seccomp-retain-set" -
$ acbuild write seccomp-retain-set-example.aci
$ acbuild end
```

Una volta eseguito, verificherò che sia ping che umount siano ora funzionanti all'interno del container. Queste operazioni richiedono inoltre delle capabilities aggiuntive per poter funzionare:

```
$ sudo rkt run --interactive --insecure-options=image
seccomp-retain-set-example.aci --caps-retain=CAP_SYS_ADMIN,CAP_NET_RAW
image: using image from file stage1-coreos.aci
image: using image from file seccomp-retain-set-example.aci
image: using image from local store for image name
quay.io/coreos/alpine-sh

# whoami
root

# ping -c 1 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=41 time=24.910 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 24.910/24.910/24.910 ms

# mount | grep /proc/bus
```

```
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
# umount /proc/bus
# mount | grep /proc/bus
```

Da notare, che altre syscall non sono ancora disponibili per l'applicazione. Infatti se ad esempio, procedessimo con il tentativo di impostare l'ora, questo provocherà un errore dovuto all'invocazione di syscall non licenziate:

```
$ sudo rkt run --interactive --insecure-options=image
seccomp-retain-set-example.aci
image: using image from file stage1-coreos.aci
image: using image from file seccomp-retain-set-example.aci
image: using image from local store for image name
quay.io/coreos/alpine-sh

# whoami
root

# adjtimex -f 0
Bad system call
```

Tipicamente, i filtri Seccomp sono definiti durante la creazione di immagini, in quanto sono strettamente legati a specifici requisiti per le applicazioni. Tuttavia, i consumers di immagini potrebbero avere bisogno di aggiustare ulteriormente o di limitare l'insieme dei sistemi di comunicazione disponibili in specifici scenari locali. Ciò può essere effettuato sia mediante l'annullamento permanente del manifest di immagini specifiche, sia tramite l'esclusione degli isolatori seccomp con opzioni di riga di comando.

I manifest di immagine, possono quindi essere manipolati manualmente, eseguendo l'unpacking dell'immagine e modificando il file manifest, oppure con strumenti di aiuto come actool. Per sovrascrivere un set di syscall predefinito per un'immagine, è quindi necessario sostituire gli isolatori seccomp esistenti nell'immagine con dei nuovi isolatori che definiscono le syscall desiderate.

Il subcomando di patch-manifest per actool manipola i set di syscall definite in un'immagine. Le opzioni possono essere utilizzate insieme per sovrascrivere i filtri seccomp specificando una nuova modalità (mantenere o reimpostare), un errno opzionale facoltativo e un insieme di syscall per filtrare. Questi comandi prendono un'immagine di ingresso, modificano gli isolatori seccomp esistenti e scrivono le modifiche ad un'immagine di uscita, come mostrato di seguito:

```
$ actool cat-manifest seccomp-retain-set-example.aci
...
    "isolators": [
      {
      "name": "os/linux/seccomp-retain-set",
        "value": {
          "set": [
            "@rkt/default-whitelist",
            "umount",
            "umount2"
          ]
        }
      }
    ]
...

$ actool patch-manifest -seccomp-mode=retain,errno=ENOSYS
-seccomp-set=@rkt/default-whitelist seccomp-retain-set-example.aci
 seccomp-retain-set-patched.aci

$ actool cat-manifest seccomp-retain-set-patched.aci
```

```
...
    "isolators": [
      {
        "name": "os/linux/seccomp-retain-set",
        "value": {
          "set": [
            "@rkt/default-whitelist",
          ],
          "errno": "ENOSYS"
        }
      }
    ]
...
```

Di seguito eseguirò l'immagine per verificare che la syscall umount(2) non sia più consentita e che venga restituito un errore personalizzato.

```
$ sudo rkt run --interactive --insecure-options=image
seccomp-retain-set-patched.aci
image: using image from file stage1-coreos.aci
image: using image from file seccomp-retain-set-patched.aci
image: using image from local store for image name
quay.io/coreos/alpine-sh

/ # mount | grep /proc/bus
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
/ # umount /proc/bus/
umount: can't umount /proc/bus: Function not implemented
```

I filtri Seccomp possono anche essere sovrascritti in fase di run-time dalla riga di comando, senza la necessità di dover modificare le immagini eseguite. L'opzione -seccomp all'esecuzione di rkt può manipolare entrambi gli isolatori "retain" e "remove" .

L'isolatore sovrascritto dalla riga di comando sostituirà tutte le impostazioni seccomp nel manifest immagine e potrà essere specificato come mostrato nell'esempio di seguito.

```
$ sudo rkt run --interactive quay.io/coreos/alpine-sh
--seccomp mode=remove,errno=ENOTSUP,socket
image: using image from file /usr/local/bin/stage1-coreos.aci
image: using image from local store for image name
quay.io/coreos/alpine-sh

# whoami
root

# ping -c 1 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: can't create raw socket: Not supported
```

Gli isolatori Seccomp sono le voci di configurazione specifiche per l'applicazione e, in una riga di comando di rkt run, devono seguire l'immagine del container dell'applicazione a cui si applicano. Inoltre, ogni applicazione all'interno di un pod può avere diversi filtri seccomp.

Come per la maggior parte delle funzioni di sicurezza, gli isolatori seccomp potrebbero richiedere una certa sintonizzazione specifica per l'applicazione al fine di poter garantire il massimo delle loro funzionalità di sicurezza. Per questo motivo, per ambienti sensibili alla sicurezza è meglio disporre di un set ben definito di requisiti di syscall ed è altrettanto importante seguire alcune linee guida come:

Permettere solo i sistemi di accesso richiesti da un'applicazione, in base al suo uso tipico.

Anche se possibile disattivare completamente il filtro seccomp, in particolar modo nei casi in cui questo risulta raramente necessario, dovrebbe comunque essere evitato.

Evitare di concedere l'accesso a sistemi pericolosi o non fidati. Ad esempio, mount(2) e ptrace(2) sono abusati tipicamente per sfuggire ai container.

Preferire un approccio whitelisting, cercando di mantenere il "keep-set" il più piccolo possibile.

4.3 AppArmor

Docker fornisce la possibilità di utilizzare AppAmror con i conatainer in esecuzione nelle istanze del sistema operativo. Per un determinato container infatti, è possibile applicare il profilo di protezione AppArmor predefinito fornito con Docker o un qualunque profilo di protezione personalizzato.

Quando si avvia un container con Docker, il sistema applica automaticamente il profilo di protezione predefinito per AppArmor. docker-default è definito dal file in /etc/apparmor.d/docker il quale nega la lettura, scrittura e l'accesso ai vari file presenti nella directory / proc.

Il seguente comando di esempio esegue un container con il profilo di protezione AppArmor predefinito docker:

```
run docker --rm -it debian: jessie bash -i
```

Per testare il profilo di protezione predefinito docker, andro a leggere il file /proc/sysrq-trigger con il comando cat.

```
root @ 88cef496c1a5: / # cat / proc / sysrq-trigger
```

L'output contiene come errore un "Permission Denied" .

```
cat: / proc / sysrq-trigger: Permission denied
```

Se il processo richiede un profilo di sicurezza diverso da quello predefinito, è possibile scrivere il proprio profilo personalizzato. Per utilizzare un profilo personalizzato, è necessario creare il file di profilo per poi in seguito caricarlo in AppArmor.

Ad esempio, ho supposto un profilo di protezione che impedisse tutto il traffico di rete. Il seguente script crea un file per un profilo di protezione denominato no-ping in /etc/apparmor.d/no_raw_net:

```
cat > /etc/apparmord.d/no_raw_net <<EOF
#include <tunables/global>

profile no-ping flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>

  network inet tcp,
  network inet udp,
  network inet icmp,

  deny network raw,
  deny network packet,
  file,
  mount,
}
EOF
```

Una volta creato il file di profilo di protezione, utilizzerò apparmor_parser per caricare il profilo in AppArmor:

```
/ sbin / apparmor_parser --replace --write-cache /etc/apparmor.d/
no_raw_net
```

Una volta caricato tale profilo, eseguirò un test per verificare che il tutto funzioni correttamente. Per applicare il profilo di protezione differente, utilizzerò l'opzione da riga di comando apparmor = ¡profilo-nome¿ al momento di mandare in esecuzione il container.

Il comando seguente crea un container con il profilo di protezione no-ping e tenta di eseguire un ping dal container.

```
$ docker run --rm -i --security-opt apparmor=no-ping debian:jessie
 ping -c3 8.8.8.8
```

Il profilo di protezione disattiva il traffico, causando un errore come quello riportato di seguito.

```
ping: Lacking privilege for raw socket.
```

Per essere sicuri che il proprio profilo personalizzato di protezione sia sempre presente e rimanga persistente nei riavvii, è possibile utilizzare cloud-init per installare il profilo in /etc/apparmor.d. A tal scopo, aggiungerò il comando di cloud-config ai metadati dell'istanza come valore del codice utente-dati.

Il seguente script di cloud-config aggiunge il profilo no-ping a /etc/apparmor.d:

```
#cloud-configs

write_files:
- path: /etc/apparmor.d/no_raw_net
  permissions: 0644
  owner: root
  content: |
    #include <tunables/global>

    profile no-ping flags=(attach_disconnected,mediate_deleted) {
      #include <abstractions/base>

      network inet tcp,
      network inet udp,
      network inet icmp,

      deny network raw,
      deny network packet,
      file,
      mount,
    }
```

E' inoltre necessario garantire che il file di servizio carichi il profilo personalizzato in AppArmor e dica a Docker di utilizzarlo. Con i comandi successivi farò quanto spiegato ora.

```
ExecStartPre = / sbin / apparmor_parser -r -W /etc/apparmor.d/no_raw_net
ExecStart = / usr / bin / run docker --security-opt apparmor = no-ping...
```

Dopo aver eseguito questi comandi, potrò riavviare l'istanza ed eseguire il container confinato dal profilo AppArmor personalizzato.

È inoltre possibile specificare con l'opzione apparmor ce il container deve essere eseguito senza alcun profilo di protezione.

```
docker run --rm -it --security-opt apparmor=unconfined
debian:jessie bash -i
```

E' inoltre possibile verificare se un determinato processo in esecuzione all'interno di un container Docker, abbia o meno il profilo di sicurezza AppArmor abilitato e si sia il profilo di default o quello personalizzato. Nel mio caso ho eseguito il controllo su 2 processi i quali PID erano rispettivamente 1927 e 2001.

```
# cat /proc/1927/attr/current
docker-default (enforce)
```

Questo primo risultato mostra che il processo PID:1927 è in esecuzione con il profilo di sicurezza di default.

```
# cat /proc/2001/attr/current
unconfined
```

Questo secondo risultato mostra che il processo PID:2001 è in esecuzione senza alcun profilo di sicurezza AppArmor.

4.4 SELinux

Come descritto nella sezione 3.1.8 , SElinux è un'architettura di sistema in cui ogni processo, ogni oggetto file o directory del sistema operativo hanno una propria etichetta. Inoltre, tali etichette possono essere assegnate anche alle porte di rete, ai dispositivi e ai nomi di host.

Se ad esempio si volessero scrivere un insieme di regole per effettuare il controllo di accesso di un'etichetta associata ad un processo, assieme ad un'altra etichetta associata ad uno specifico oggetto, come ad esempio un file, possiamo farlo attraverso alle policy SELinux nella quale il kernel del sistema operativo, imporrà delle regole di tipo MAC.

C'è da aggiungere, che solitamente il proprietario di un oggetto non ha discrezione sugli attributi di protezione di un oggetto. Infatti viene applicato lo standard di controllo di accesso Linux (flag di proprietario / gruppo + permesso come rwx) o più comunemente DAC.

Quindi con SELinux tutto può essere controllato attraverso l'assegnazione di etichette è spesso chiamato controllo ad accesso discrezionale come descritto nella sezione 3.1.4. Inoltre, SELinux non ha alcun concetto di UID o di proprietà dei file.

Inoltre, SELinux non consente di eseguire i passaggi di fase DAC, poichè SELinux è un modello di esecuzione parallelo. Dunque, un'applicazione deve essere consentita sia da SELinux che DAC perchè questa possa eseguire determinate attività. Questo spesso porta in confusione diversi amministratori di sistemi linux, poichè quando un processo viene rifiutato, o che i suoi permessi vengono negati, si pensa che ci sia qualche problema relativo al DAC, quando invece il problema potrebbe essere dovuto all'assegnazione di etichette SELinux.

Prima di applicare SELinux alle varie tecnologie di container management, inizierò con l'utilizzo di SELinux su una distribuzione Linux installata su una macchina host, in modo da poter comprendere meglio i vari principi e meccanismi di funzionamento di questo Linux Security Module, per poi arrivare in seguito ad utilizzare tale componente per la gestione della sicurezza dei container. La distribuzione installata sulla macchina, ai fini di tale dimostrazione è CentOS versione 7. Come introduzione cercherò di spiegare in modo pratico il funzionamento del DAC, per poi arrivare al livello superiore MAC, dove viene implementato SELinux.

Considero quindi un modello di sicurezza tradizionale basato su DAC, dove si hanno tre entità: User, Group e Other, i quali possono avere una combinazione di permessi di lettura, scrittura ed esecuzione (r, w, x) su uno specifico file o directory. Se ad esempio, un utente Alias crea un file nella sua home directory, quell'utente avrà accesso in lettura ed in scrittura, così come lo avrebbe anche un Alias Group. L'entità "Other" invece, potrebbe non avere accesso ad essa.

```
ls -l /home/Alias/

#Output:
total 4
-rwxrw-r--. 1 Alias Alias 41 Aug 6 22:45 myscript.sh
```

L'utente Alias potrà quindi concedere o limitare l'accesso a questo file, ad altri utenti e gruppi, o cambiare il proprietario di tale file. Queste azioni possono lasciare esposti i file critici agli account che non hanno bisogno di questo accesso. Alias potrà inoltre essere più sicuro, ma questo è discrezionale, infatti non è possibile per l'amministratore di sistema applicarlo per ogni singolo file nel sistema.

Ora considero un caso deve, quando viene eseguito un processo Linux, esso, potrà essere mandato in esecuzione come utente root o con un altro account che è in possesso dei privilegi di superuser. Ciò significa che se un utente malevolo, riuscisse nell'intento di prendere il controllo dell'applicazione, può utilizzare quell'applicazione per ottenere l'accesso a qualsiasi risorsa a cui l'account utente abbia accesso. Quindi, per i processi che sono in esecuzione con privilegi di root, significherebbe, che tali processi avrebbero accesso all'intero sistema.

Si può infatti pensare ad uno scenario in cui si vuole impedire agli utenti di eseguire degli script di shell dalle loro home directory. Questo può accadere quando gli sviluppatori lavorano su un sistema di produzione. Infatti, si potrebbe volere che essi visualizzassero i file di registro, ma che tali sviluppatori, non siano abilitati ad utilizzare i comandi su o sudo e non si vorrebbe inoltre che essi possano eseguire script dalle loro directory home. E' a questo punto che si decide di utilizzare il modulo di sicurezza di SELinux allo scopo di poter mettere a punto i requisiti necessari al controllo degli accessi. A questo punto quindi, sarà possibile, grazie all'utilizzo di SELinux, definire cosa un utente può o meno fare e lo stesso vale anche per un dato processo, confinando quest' ultimo, come ogni altro processo, nel proprio dominio in modo tale che il processo possa interagire solo con determinati tipi di file e con altri processi dai domini consentiti. Questo accorgimento, eviterà che un utente malintenzionato, possa dirottare qualsiasi processo allo scopo di ottenere l'accesso dell'intero sistema host.

Ora costruirò un server di test allo scopo di spiegare il funzionamento di SELinux. Questo server manderà in esecuzione sia un server Web che un server SFTP. Dopo aver installato il demone Apache e vsftp, installerò i pacchetti necessari all' utilizzo di SELinux, allo scopo di poter garantire che gli ultimi comandi SELinux siano opportunamente supportati.

Con i comandi *'yum install httpd'* e *'yum install vsftpd'* eseguirò l'installazione di demone Apache e vsftp.

Invece con l'utilizzo di *'service httpd start'* e *'service vsftpd start'* manderò in esecuzione queste componenti.

Come passo successivo mi occuperò di installare i packages di SELinux. Di seguito elencherò una lista di quelli principalmente utilizzati. Da notare che alcuni di questi sono già installati di default, quindi con il comando *'rpm -qa — grep selinux'* potrò verificare quale di questi sono installati così da poter eventualmente installare quelli mancanti con il comando *'yum install package_name'*.

- policycoreutils (fornisce utility per la gestione di SELinux)
- policycoreutils-python (fornisce utility per la gestione di SELinux)
- selinux-policy (fornisce la politica di riferimento di SELinux)
- selinux-policy-targeted (fornisce la politica mirata di SELinux)
- libselinux-utils (fornisce alcuni strumenti per la gestione di SELinux)
- setroubleshoot-server (fornisce strumenti per decifrare i messaggi del registro di controllo)
- setols (fornisce strumenti per il monitoraggio del log di controllo, criteri di query e gestione del contesto dei file)
- setools-console (fornisce strumenti per il monitoraggio del registro di controllo, criteri di query e gestione del contesto dei file)
- mcstrans (strumenti per tradurre diversi livelli in un formato facile da capire)

A questo punto il sistema è caricato con tutti i pacchetti SELinux ed il server Apache e SFTP sono in esecuzione con le loro configurazioni di default. Sarà quindi possibile iniziare con il configurare la modalità di esecuzione di SELinux, che può essere di tre tipi:

- Enforcing
- Permissive
- Disabled

In modalità enforcing, SELinux applicherà la propria politica sul sistema Linux e si assicurerà che qualsiasi tentativo di accesso non autorizzato da parte di utenti e processi venga negato.

La modalità permissive è come uno stato di semi-abilitato. Infatti SELinux in questa modalità non applicherà alcuna politica e quindi non verrà negato alcun accesso, anche se ogni tipo di violazione delle norme di sicurezza, verrà comunque registrata nei registri di controllo.

Infine, la modalità Disabled è autoesplicativa, ovvero il sistema non funzionerà con l'ausilio della sicurezza avanzata fornita mediante la configurazione di policy SELinux.

Per verificare la modalità in cui è in esecuzione SELinux si potrà utilizzare il comando *'getenforce'* oppure il comando *'setstatus'*.

Con il comando seguente sarà possibile vedere il file di configurazione completo.

```
cat /etc/selinux/config
```

Nel mio caso l'output è risultato il seguente:

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
# enforcing - SELinux security policy is enforced.
# permissive - SELinux prints warnings instead of enforcing.
# disabled - No SELinux policy is loaded.
SELINUX=disabled
# SELINUXTYPE= can take one of these two values:
# targeted - Targeted processes are protected,
# minimum - Modification of targeted policy.
# mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

Si noti che in questo file sono presenti due direttive. La direttiva SELINUX detta la modalità SELinux e può avere tre valori possibili come discusso precedentemente. La seconda direttiva SELINUXTYPE, determina invece la politica che verrà utilizzata. Il valore predefinito è targeted. Con una politica di tipo targeted, SELinux consentirà di personalizzare e mettere a punto le autorizzazioni di controllo degli accessi. L'altro valore possibile sarà "MLS" 3.1.8. La modalità di utilizzo potrà essere quindi modificata, andando a modificare il file di configurazione, con consecutivo riavvio del sistema. In seguito al processo di riavvio del sistema, Isarà possibile vedere tutti i file nel server etichettati con un contesto SELinux. Un altro modo per poter settare la modalità di esecuzione in SELinux senza dover andare ad editare il file di configurazione, è quello di passare al comando *'setenforce'* l'attributo che in questo caso sarà enforcing.

Il parte principale di SELinux è basata sulla definizione di policy di sicurezza. Una policy infatti, è ciò che il nome stesso implica, ovvero un insieme di regole che definiscono la sicurezza e i diritti di accesso per tutto ciò che è presente nel sistema. Ovvero, quando vengono portate a compimento tutte le decisioni di sicurezza intese come la definizione di utenti, ruoli, processi e file, per poi collegare l'una all'altra ciascuna di queste entità.

Per capire meglio cosa è e come funziona una policy, descriverò di seguito i termini di base che la compongono, per poi in un secondo momento, entrare maggiormente nel dettaglio con degli esempi applicativi. Una policy SELinux quindi, definisce l'accesso degli utenti ai ruoli, l'accesso di questi ruoli a dei domini e l'accesso ai tipi di dominio.

Users
SELinux ha una serie di utenti predefiniti ed ogni normale account utente Linux sarà mappato ad uno o più di questi utenti SELinux.

In Linux, un utente esegue un processo. Questo può essere semplice come l'utente Alias che apre un documento nell'editor vi (sarà l'account di Alias che esegue il processo vi) o un account di servizio che esegue il daemon httpd. In SELinux invece, un processo come ad esempio un demone o un programma in esecuzione, verrà chiamato definito come subject.
Roles
Un ruolo è come un gateway che si trova tra un utente e un processo. Esso ha lo scopo di definire quali utenti possono accedere a tale processo. I ruoli non sono considerati come gruppi, ma più come dei filtri, infatti un utente può entrare o assumere un ruolo in qualsiasi momento, ma a condizione che il ruolo lo conceda. La definizione di un ruolo nella politica SELinux definisce quindi quali utenti hanno accesso a quello specifico ruolo, definendo inoltre a quali domini di processo il ruolo stesso avrà accesso. I ruoli entrano in gioco poichè parte di SELinux implementa il cosiddetto *Role Based Access Control* (RBAC) [46].
Subjects and Objects
Un soggetto è un processo e tale processo, può potenzialmente influenzare un oggetto. Un oggetto invece, in SELinux è tutto ciò su cui si può agire. Infatti può trattarsi di un file, una directory, una porta, un socket TCP, il cursore ecc.. Le azioni che un soggetto può eseguire su un oggetto sono definite dalle autorizzazioni del soggetto.
Domain
Un dominio è il contesto in cui può essere eseguito un oggetto (processo) SELinux. Quel contesto è come un involucro attorno all'argomento, il quale avrà lo scopo di indicare a tale processo, cosa questo potrà o non potrà fare. Ad esempio, il dominio definirà quali file, directory, collegamenti, dispositivi o porte sono accessibili all'argomento.
Type
Un tipo è la circostanza che viene definita per il contesto di un file e che ne stabilisce lo scopo del file. Ad esempio, il contesto di un file potrebbe dettare che si tratta di una pagina Web o che il file appartiene alla directory /etc o che il proprietario del file è un utente SELinux specifico.

Riepilogando, una policy SELinux definirà quindi l'accesso degli utenti ai ruoli, l'accesso ai ruoli ai domini e l'accesso ai tipi di dominio. Per prima cosa l'utente dovrà essere autorizzato a inserire un ruolo, per cui tale ruolo dovrà essere autorizzato ad accedere ad uno specifico dominio. Il dominio, a sua volta, sarà limitato a poter accedere solo a determinati tipi di file. Inoltre, come detto precedentemente, tali policy in SELinux sono 'targeted'di default. Questo significa che, come impostazione di default, SELinux limiterà nel sistema, solo determinati processi. Qui processi che invece non saranno definiti come targeted, verranno eseguiti in unconfined domains. Un'alternativa può essere data dalla definizione di un modello 'deny-by-default', in cui ogni accesso è negato a meno che non sia approvato dalla policy stessa. Questa risulta essere un'implementazione molto più sicura rispetto alla precedente, ma ciò significherebbe anche che gli sviluppatori devrebbero anticipare ogni singola possibile autorizzazione che ogni singolo processo potrebbe richiedere su ogni singolo oggetto possibile.

La policy SELinux inoltre, non sostituisce la tradizionale sicurezza DAC. Infatti se una regola DAC proibisce a un utente l'accesso a un file, le regole dei criteri SELinux non saranno nemmeno valutate, perché la prima linea di difesa avrà già bloccato l'accesso. Quindi le decisioni di sicurezza di SELinux entrano in gioco dopo che la sicurezza del DAC è stata valutata.

Quando viene avviato un sistema abilitato per SELinux, il criterio viene caricato in memoria. La policy arriverà in un formato modulare, proprio come i moduli del kernel che vengono caricati all'avvio, ed è proprio come tali moduli, che possono essere aggiunti e rimossi dinamicamente dalla memoria in fase di esecuzione. Il policy store utilizzato da SELinux sarà la componente che si occuperà di tener traccia dei moduli che sono stati caricati e con il comando *'sestatus'* sarà possibile mostrare il nome dell'archivio delle policy. Il comando *'semodule -l'* invece, elencherà tutti i moduli di policy SELinux caricati attualmente in memoria.

I file di policy avranno un'estensione .pp e per CentOS 7, con il comando scritto di seguito, sarà possibile visualizzare tutte le policy SELinux definite nel proprio sistema.

```
ls -l /etc/selinux/targeted/modules/active/modules/
```

Invece con il comando successivo, sarà possibile vedere le policy SELinux che in questo caso sono attive nel proprio sistema.

```
ls -l /etc/selinux/targeted/policy/
```

Lo scopo di SELinux è quindi quello di garantire il modo in cui i processi accedono ai file in uno specifico ambiente Linux. Senza SELinux infatti, un processo o un'applicazione come il daemon Apache verrà eseguito nel contesto dell'utente che l'ha avviato. Quindi, se il proprio sistema venisse compromesso da un'applicazione malevola in esecuzione con l'utente root, tale applicazione, potrà fare tutto ciò che vorrà sull'intero sistema e su tutti i file presenti in esso. Quindi SELinux farà in modo che un processo o un'applicazione abbiano solo i diritti di cui hanno realmente bisogno per funzionare e nulla di più. La policy SELinux per l'applicazione determinerà a quali tipi di file si ha bisogno di accedere e quali processi potranno essere eseguiti da tale transizione. Tali policy sono dunque scritte dagli sviluppatori di applicazioni e fornite con la distribuzione Linux che le supporta.

La prima parte del meccanismo di sicurezza di SELinux, ha lo scopo di etichettare ogni entità nel sistema Linux. Un'etichetta infatti, è come qualsiasi altro attributo di file o processo (proprietario, gruppo, data di creazione, ecc..) che mostra il contesto di una determinata risorsa. Per contesto si vuole intendere una raccolta di informazioni relative alla sicurezza, le quali aiuteranno SELinux a prendere decisioni di controllo degli accessi. Tutto in un sistema Linux può avere un contesto di sicurezza come ad esempio un account utente, un file, una directory, un demone o una porta possono avere tutti i loro contesti di sicurezza. Tuttavia, il contesto di sicurezza significherà cose diverse per diversi tipi di oggetti.

Se ad esempio verifichiamo l'output di un normale comando ls -l relativo alla directory /etc, potremmo notare quanto segue.

```
ls -l /etc/*.conf

...
-rw-r--r--. 1 root root 19 Aug 19 21:42 /etc/locale.conf
-rw-r--r--. 1 root root 662 Jul 31 2013 /etc/logrotate.conf
-rw-r--r--. 1 root root 5171 Jun 10 07:35 /etc/man_db.conf
-rw-r--r--. 1 root root 936 Jun 10 05:59 /etc/mke2fs.conf
...
```

Utilizzando sempre lo stesso comando, ma con -Z al posto di -l, l'output risulterà leggermente differente:

```
...
-rw-r--r--. root root system_u:object_r:locale_t:s0 /etc/locale.conf
-rw-r--r--. root root system_u:object_r:etc_t:s0 /etc/logrotate.conf
-rw-r--r--. root root system_u:object_r:etc_t:s0 /etc/man_db.conf
-rw-r--r--. root root system_u:object_r:etc_t:s0 /etc/mke2fs.conf
...
```

Ora infatti disponiamo di una colonna aggiuntiva di informazioni dopo la proprietà dell'utente e del gruppo. Questa colonna mostra i contesti di sicurezza dei file.

Adesso prendendo in considerazione ad esempio la seconda riga, quella corrispondente a logrotate.conf potrò identificare la colonna dove è indicato **'system_u:object_r:etc_t:s0'** come la parte relativa al security context di SELinux. Tale contesto di sicurezza si divide in quattro sottoparti, di cui ognua di esse è separata dai due punti (:). La prima parte *system_u* indica il contesto utente SELinux per il file. Questo perchè ogni account utente Linux viene associato ad un utente SELinux e in questo caso l'utente root proprietario del file verrà mappato all'utente system_u SELinux. Questa mappatura viene eseguita dalla policy di SELinux.

La seconda parte invece, specifica il ruolo di SELinux, che in questo caso è definito come object_r.

La terza parte, indica il tipo di file, che in questo caso è definito come etc_t. Questa parte infatti, definisce il tipo a cui appartiene un file o una directory. Ipoteticamente, è possibile pensare

tipo come ad una sorta di "gruppo"o attributo per il file, ovvero come ad un modo di classificare tale file.

Inizialmente avevo installato e mandato in esecuzione sulla macchina host i due servizi Apache e SFTPD. Ora, utilizzando il comando indicato di seguito, potrò visualizzare i processi Apache ed SFTPD in esecuzione, oltre a visualizzare il SELinux context di ogniuno di questi processi.

```
ps -efZ | grep 'httpd\|vsftpd'

system_u:system_r:httpd_t:s0 root 7126 1
0 16:50 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0 apache 7127 7126
0 16:50 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0 apache 7128 7126
0 16:50 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0 apache 7129 7126
0 16:50 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0 apache 7130 7126
 0 16:50 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0 apache 7131 7126
0 16:50 ? 00:00:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:ftpd_t:s0-s0:c0.c1023 root 7209
 1 0 16:54 ? 00:00:00
/usr/sbin/vsftpd /etc/vsftpd/vsftpd.conf
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 root 7252 2636
 0 16:57 pts/0 00:00:00 grep --color=auto httpd\|vsftpd
```

L'output di cui sopra, mostra i vari contesti di sicurezza associati ad ogni processo. *User*, *Roles* e *sensitivity* funzionano esattamente come gli stessi contesti per i file. Il dominio invece è unico per i processi. E' quindi possibile vedere che alcuni processi sono in esecuzione all'interno del dominio *httpd_t*, mentre uno è in esecuzione nel dominio *ftpd_t*.

Quindi, il dominio, avrà lo scopo di fornire al processo un contesto il quale implicherà che tale processo possa essere eseguito solo al suo interno. Quindi un dominio indica cosa un processo in esecuzione al suo interno possa o meno fare. Questo tipo di confinamento infatti, garantisce che ciascun dominio del processo possa agire solo su determinati tipi di file e nulla più.

Utilizzando questo modello, anche se un processo viene dirottato da un altro processo o da un utente malevolo, il peggio che questo potrebbe fare è danneggiare i file a cui ha accesso, come ad esempio il *demone vsftp* non potrà avere accesso ai file usati da *Sendmail* o *Samba*. Questo tipo di restrizione è implementata dal livello del kernel, è applicata in quanto la politica di SELinux viene caricata in memoria e pertanto il controllo dell'accesso diventa obbligatorio.

Finora ho descritto come, sia i file che i processi possono avere contesti differenti e che sono limitati ai loro tipi o domini. Adesso invece, invece spiegherò come funziona un processo. Un processo, per essere eseguito deve poter accedere ai suoi file ed eseguire alcune azioni su di essi come ad esempio aprire, leggere, modificare o eseguire. Ogni processo inoltre, può avere accesso solo a determinati tipi di risorse come file, directory, porte, ecc..

SELinux quindi, come detto precedentemente, stipula queste regole di accesso in una policy. Le regole di accesso seguono a loro volta una struttura di istruzioni e permessi standard come descritto di seguito:

```
allow <domain> <type>:<class> { <permissions> };
```

Nella riga di cui sopra abbiamo *domain* e *type* di cui ho già parlato. *Class* invece, definisce cosa rappresenta effettivamente la risorsa, ovvero: file, directory, collegamento simbolico, dispositivo, porte, cursore ecc.. e come ultimo abbiamo l'attributo *permissions*. Questa riga quindi vuole definire che se un processo è appartenente ad un certo dominio e l'oggetto risorsa a cui sta tentando di accedere è di tipo e classe certa, si può decidere se consentire o negare l'accesso a tale risorsa.

Per vedere in modo più chiaro come funziona, descriverò un esempio pratico prendendo in considerazione i contesti di sicurezza del *demone httpd* in esecuzione sul sistema.

La directory home di default per il server web è /var/www/html. Ora creerò un file all'interno di questa directory ed in seguito controllerò il suo contesto. e descritto di seguito:

```
touch /var/www/html/index.html

ls -Z /var/www/html/*

#output:
-rw-r--r--. root root unconfined_u:object_r:
httpd_sys_content_t:s0 /var/www/html/index.html
```

Il contesto del file per i contenuti web sarà quindi *httpdsyscontent_t*:

Ora utilizzerò il comando sesearch per verificare il tipo di accesso consentito per il *demone httpd*.

```
sesearch --allow --source httpd_t --target httpd_sys_content_t --class file
```

I flag usati con il comando sono abbastanza autoesplicativi: il dominio di origine è httpd_t, ovvero lo stesso dominio in cui *Apache* è in esecuzione.

```
Found 4 semantic av rules:
allow httpd_t httpd_sys_content_t : file { ioctl read getattr lock open } ;
allow httpd_t httpd_content_type : file { ioctl read getattr lock open } ;
allow httpd_t httpd_content_type : file { ioctl read getattr lock open } ;
allow httpd_t httpdcontent : file { ioctl read write create
getattr setattr lock append unlink link rename execute open } ;
```

La prima linea indica che il *demone httpd* (il server Web Apache) ha il controllo I/O, read, get attribute, lock, e open sui file del tipo httpdsys_content_type. In questo caso il nostro file index.html ha lo stesso tipo. Di seguito, cambierò il permesso della cartella /var/www/ e del suo contenuto, seguito da un riavvio del *daemone httpd*.

```
chmod -R 755 /var/www

service httpd restart
```

Provando ad accedere a tale bagina da un qualunque browser, avrò come risultato la stampa a video "Hello World" definita all'interno del file html creato precedentemente, il che vuol dire che tutto è andato a buon fine. Infatti, il demone httpd è autorizzato ad accedere ad un particolare tipo di file. Adesso invece, cambiando il contesto del file con il comando *chcon*.

```
chcon --type var_t /var/www/html/index.html
```

Il flag *–type* per il comando consente di specificare un nuovo tipo per la risorsa di destinazione, che in questo caso è stato modificato nel tipo *var_t*. Successivamente, per avere conferma dell'avvenuta modifica ho eseguito il comando di seguito.

```
ls -Z /var/www/html/

#Output:
-rwxr-xr-x. root root unconfined_u:object_r:var_t:s0 index.html
```

A questo punto, quando si proverà nuovamente ad accedere alla pagina Web, il *daemone httpd* tenterà di leggere il file, ma questo non sarà possibile e verrà restituito il seguente errore:

```
#Output
Accesso alla pagina Web con impostazioni SELinux errate
```

Questo accade perchè, per quanto riguarda SELinux il server Web è autorizzato ad accedere solo a determinati tipi di file e *var_t* non è uno dei tipi contesti. Infatti, poiché ho modificato il contesto del file index.html in *var_t*, Apache non è più in grado di leggerlo e verrà restituito un errore. Per ripristinare il corretto funzionamento ho utilizzato il comando *restorecon*.

```
restorecon reset /var/www/html/index.html context
unconfined_u:object_r:var_t:s0->unconfined_u:object_r:
httpd_sys_content_t:s0
```

Riguardo alla parte relativa al contesto o *type*, definito in SELinux, questo è l'applicazione di qualcosa che si può definire come "eredità del contesto" . Questo significa che, se non specificato dalla politica, i processi e i file verranno creati con i contesti dei loro genitori.

Quindi, se abbiamo un processo "processo1"che genera un altro processo figlio chiamato "processo2" , il processo generato verrà eseguito di default, nello stesso dominio del "processo1"se non diversamente specificato dalla policy di SELinux.

Allo stesso modo, se abbiamo una directory con un tipo di contesto, qualsiasi file o directory creata al suo interno, avrà lo stesso contesto, a meno che la policy non indichi diversamente. Con questo comando *ls -Z /var/www* verificherò il contesto della directory /var/www/.

```
drwxr-xr-x. root root system_u:object_r:httpd_sys_script_exec_t:s0 cgi-bin
drwxr-xr-x. root root system_u:object_r:httpd_sys_content_t:s0 html
```

La directory html in /var/www/ ha il contesto di tipo httpdsyscontent_t. Infatti, come visto in precedenza, il file *index.html* al suo interno ha lo stesso contesto, ovvero il contesto del padre.

Questa ereditarietà però, non viene conservata quando i file vengono copiati in un'altra posizione. In un'operazione di copia, il file o la directory copiati assumono il tipo di contesto della posizione di destinazione, com'è possibile vedere nell' esempio riportato di seguito.

```
cp /var/www/html/index.html /var/

ls -Z /var/index.html

#output
-rwxr-xr-x. root root unconfined_u:object_r:var_t:s0 /var/index.html
```

Se si vuole quindi evitare il cambio di contesto all'atto della copia di un file in un'altra destinazione, si può fare aggiungendo il flag *–preserver = context* nel comando *cp*.

Quando invece i file o le directory vengono spostati, i contesti originali vengono mantenuti senza alcuna modifica.

Il concetto di contesto durante la copia o lo spostamento di file o directory, è importante, poichè se ad esempio si volessero copiare tutti i file HTML del proprio server web in una directory separata, allo scopo di poter semplificare il processo di backup e anche per rafforzare la sicurezza, poichè non si vuole che nessun utente malevolo riesca ad identificare con facilità dove sono situati questi determinati file. A questo punto si andrà ad aggiornare il controllo di accesso alla directory, si modificherà il file di configurazione *web* in modo che questo punti alla nuova posizione e si riavvierà il servizio. Il servizio però non funziona e restituisce un errore, questo perchè andando a vedere i contesti della directory con i relativi file di interesse, si nota che ora i contesti saranno del tipo *default_t* e non più del tipo *httpdsyscontent_t*, poichè è stata eseguita una copia di tali file e quindi il contesto non viene mantenuto. Sarà quindi necessario riportare il contesto a quello originale *httpdsyscontent_t* con il comando *restorecon -Rv /folder_name*, oppure come detto in precedenza, effettuare la copia dei file in una nova destinazione, utilizzando l'attributo *–preserver = context* nel comando *cp*. A questo punto tutto funzionerà correttamente.

Dopo aver visto come i processi accedono alle risorse del file system, ora descriverò come i processi accedono a loro volta ad altri processi. La transizione del dominio è il metodo in cui un processo cambia il suo contesto da un dominio all'altro. Infatti, se ad esempio, avessimo un processo procA in esecuzione in un contesto de tipo *context_procA_t*, con la transizione del

dominio il processo procA potrà eseguire un'applicazione ovvero un programma o uno script eseguibile chiamata ad esempio *AppScript* che genererà un altro processo. Questo nuovo processo infatti, potrebbe essere chiamato procB e potrebbe essere in esecuzione all'interno del dominio context_procB_t. In questo modo, si sta passando dal dominio di tipo context_procA_t al dominio di tipo context_procB_t, tramite l'utilizzo di *AppScript*. L'eseguibile *AppScript* funziona quindi come un entrypoint per il contesto context_procB_t.

Ora considero il processo *vsftpd* in esecuzione sul server. In seguito considero il processo *systemd*, il quale viene eseguito all'interno di un contesto di init_t. Questo risulta essere un processo di breve durata. Tale processo *systemd* avrà lo scopo di invocare l'eseguibile binario /usr/sbin/vsftpd, che ha un contesto di tipo di ftpdexect. Quando l'eseguibile binario verrà avviato, questo diventerà il demone *vsftpd* e verrà eseguito all'interno del dominio ftpd_t. Quindi il processo *systemd*, che in esecuzione nel dominio init_t, sta eseguendo un file binario identificato dal tipo ftpdexect. Questo file binario come detto di cui sopra, avvierà il demone *vsftpd* all'interno del dominio ftpd_t.

Questo tipo di transizione infatti, non è qualcosa che l'applicazione o l'utente possono controllare. Questo viene stabilito a priori nella policy SELinux che si carica in memoria all'avvio del sistema. Questo esempio in un server non SELinux, implica che un utente possa avviare un processo passando a un account di livello più alto e sempre a condizione che questo abbia il permesso di farlo. Utilizzando SELinux invece, questo tipo di accesso è controllato da policy descritte a priori. Ed è per questo motivo che SELinux è di tipo *Mandatory Access Control*.
La transizione del dominio sarà quindi soggetta a tre rigide regole:

- Il processo principale del dominio di origine deve disporre dell'autorizzazione di esecuzione per l'applicazione che si trova tra entrambi i domini e questo è considerato il punto di accesso.
- Il contesto del file per l'applicazione deve essere identificato come un punto di accesso per il dominio di destinazione.
- Il dominio originale deve essere autorizzato a passare al dominio di destinazione.

Per vedere se un dato processo come ad esempio il *vsftpd daemon*, risulta conforme a queste tre regole, è possibile utilizzare il comando *sesearch*, il quale fornisce diverse opzioni.

Innanzitutto, il dominio di origine *init_t* dovrà disporre dell'autorizzazione di esecuzione sull'applicazione entrypoint con il contesto ftpdexec_t.

```
sesearch -s init_t -t ftpd_exec_t -c file -p execute -Ad

#output
Found 1 semantic av rules:
allow init_t ftpd_exec_t : file { read getattr execute open } ;
```

Il risultato mostra che i processi all'interno del dominio init_t possono leggere, ottenere attributi, eseguire e aprire i file del contesto ftpdexec_t.

Successivamente, controllerò se il file binario è il punto di accesso per il dominio di destinazione ftpd_t.

```
sesearch -s ftpd_t -t ftpd_exec_t -c file -p entrypoint -Ad

#output
Found 1 semantic av rules:
allow ftpd_t ftpd_exec_t : file { ioctl read getattr lock execute
execute_no_trans entrypoint open } ;
```

Infine, il dominio di origine init_t dovrà disporre dell'autorizzazione per la transizione al dominio di destinazione ftpd_t, e come risulta possibile vedere di seguito, il dominio di origine possiede tale autorizzazione.

```
sesearch -s init_t -t ftpd_t -c process -p transition -Ad

#output
Found 1 semantic av rules:
allow init_t ftpd_t : process transition ;
```

SELinux inoltre identifica anche i processi che vengono eseguiti all'interno di domini non confinati identificati dal tipo di contesto *unconfined_t*. Questo tipo di dominio è specificato per gli utenti che eseguono i loro processi di default. I processi identificati da tale dominio, avrebbero quindi i diritti di accesso completo nel sistema. Ma take accesso non è arbitrario e va definito dalla policy SELinux.

Nella prima parte ho descritto la parte di una policy relativa alle regole per poi in seguito descrivere la parte relativa ai contesti. Adesso ci si focalizzerà sulla parte di SELinux relativa agli utenti. Gli utenti in SELinux, sono delle entità diverse dai normali account utente che si trovano in Linux, incluso l'account root. Questi utenti SELinux, sono definiti nella policy che verrà caricata in memoria al momento dell'avvio. I nomi degli utenti terminano con _u, proprio come i tipi di nomi di dominio terminano con _t e i ruoli terminano con _r. Diversi utenti SELinux hanno inoltre, diversi diritti nel sistema e questo è ciò che li rende utili. L'utente SELinux elencato nella prima parte del contesto di sicurezza di un file sarà quindi l'utente che possiederà quel file. Un'etichetta utente presente in un contesto di processo avrà lo scopo di mostrare il privilegio dell'utente SELinux con cui è in esecuzione il processo in questione.

Quando SELinux viene applicato, ogni account utente Linux regolare verrà a sua volta mappato su un account utente SELinux. E' importante precisare, che possono esserci più account utente mappati allo stesso utente SELinux. Questo tipo di mappatura consente a un account regolare di ereditare l'autorizzazione della sua controparte SELinux.

Per visualizzare questa tipo di mappatura, sarà necessario eseguire il comando indicato di seguito.

```
semanage login -l

#output
Login Name SELinux User MLS/MCS Range Service

__default__ unconfined_u s0-s0:c0.c1023 *
root unconfined_u s0-s0:c0.c1023 *
system_u system_u s0-s0:c0.c1023 *
```

La prima colonna dell'output di cui sopra, rappresenta gli account utente Linux locali. Inoltre, qualsiasi account utente Linux regolare viene prima associato al *Login Name* di default. Questo verrà quindi mappato all'utente SELinux che verrà identificato come unconfined_u. La terza colonna invece, mostra la classe di sicurezza (MLS/MCS) associata a ciascun utente. L'utente root invece, non viene mappato al *Login Name* di default, ma gli viene assegnata un voce propria. Anche root verrà però mappato all'utente SELinux unconfined_u.

system_u invece, è una classe di utenti differente, che è stata pensata allo scopo di poter eseguire processi o demoni.

Per vedere invece quali utenti SELinux sono disponibili nel sistema, bisognerà utilizzare il comando *semanage user -l*.

```
                Labeling MLS/ MLS/
SELinux User Prefix MCS MCS
                          Level Range SELinux Roles

guest_u user s0 s0 guest_r
root user s0 s0-s0:c0.c1023 staff_r sysadm_r
system_r unconfined_r
staff_u user s0 s0-s0:c0.c1023 staff_r sysadm_r
```

```
system_r unconfined_r
sysadm_u user s0 s0-s0:c0.c1023 sysadm_r
system_u user s0 s0-s0:c0.c1023 system_r unconfined_r
unconfined_u user s0 s0-s0:c0.c1023 system_r unconfined_r
user_u user s0 s0 user_r
xguest_u user s0 s0 xguest_r
```

Ora dall'output di cui sopra, è possibile vedere che l'utente unconfined_u è mappato ai ruoli system_r e unconfined_r. La politica SELinux consentirà quindi a questi ruoli di eseguire processi nel dominio unconfined_t. Allo stesso modo, utente sysadm_u è autorizzato per il ruolo sysadm_r. Ognuno di questi ruoli avrà quindi diversi domini a loro autorizzati.

Dal primo output, si è visto che l'accesso di default viene mappato come unconfined_u , proprio come l'utente root viene mappato all'utente unconfined_u di SELinux. Dato che l'accesso _default_rappresenta qualsiasi normale account utente Linux, questi account saranno autorizzati anche per i ruoli system_r e unconfined_r.

Questo vuole significare che qualsiasi utente Linux mappato all'utente unconfined_u avrà i privilegi per avviare qualsiasi applicazione che verrà eseguita all'interno del dominio unconfined_t.

Per avere una visione più chiara di come SELinux impone la sicurezza per gli account utente, descriverò un esempio pratico. Innanzitutto prenderò in considerazione un normale account utente. Come amministratore di sistema, l'utente avrà quindi gli stessi privilegi SELinux illimitati come l'account di root. Nell'esempio che seguirà, proverò a cambiare tali privilegi, imponendo dei vincoli che non permettono ad un determinato utente, di passare ad altri account, incluso l'account di root.

Per prima cosa controllerò la possibilità dell'utente di passare a un altro account.

```
[regularuser@localhost ~]$ su - switcheduser
Password:
[switcheduser@localhost ~]$
```

Nell'output di cui sopra, presupponendo che di conoscere la password, l'utente normale passa ad un altro account utente.

Successivamente cambierò il mapping dell'utente regularuser di SELinux a utente_u.

```
semanage login -a -s user_u regularuser
```

La modifica non avrà effetto fino a quando non verrà disconnesso l'utente identificato come *regularuser*. Dopo la disconnessione con successivo tentativo di riconnessione come utente *regularuser*, vedrò il messaggio indicato di seguito.

```
su: Authentication failure
```

Questa procedura può essere utile se il sistema è gestito da più persone fisiche e si vuole limitare la loro capacità di cambiare account. Questo però non implica il fatto che loro possano accedere direttamente come utenti con privilegi elevati. Un altro esempio può essere quello di limitare all'account guest_t di eseguire script dalle proprie directory home. Eseguendo il comando *getsebool* sarà possibile verificare il valore booleano assegnato.

```
getsebool allow_guest_exec_content

#output
guest_exec_content --> on
```

A questo punto, con il comando di seguito assegnerò all' utente guestuser il mapping a guest_u

```
semanage login -a -s guest_u guestuser
```

Successivamente, effettuerò l'accesso come *guestuser* e manderò in esecuzione un semplice script il quale mi restituirà il seguente output:

```
[guestuser@localhost ~]$ ~/myscript.sh

#output
This is a test script
```

Ora cambierò il boleano di allow_guest_exec_content ad off con il comando *setsebool*. Dopo tale modifica, non sarà più possibile eseguire lo script e mi verrà restituito l'errore indicato di seguito.

```
-bash: /home/guestuser/myscript.sh: Permission denied
```

Adesso invece, descriverò come utilizzare i ruoli allo scopo di poter limitare l'accesso degli utenti. Come detto in precedenza, un ruolo in SELinux si trova tra l'utente e il dominio del processo e controlla in quali domini può entrare il processo dell'utente. Nel contesto di sicurezza relativo ai file, i ruoli vengono considerati con un valore generico identificato come object_r. Nel contesto dei processi associati ai relativi utenti, si presentano invece delle differenze.

Accederò quindi con un altro account utente "restricteduser" , il quale è un account che ho creato appositamente con delle restrizioni SELinux. Con il comando seguente potrò vedere com'è configurato tale account per SELinux.

```
[restricteduser @ localhost ~] $ id -Z

#output
unconfined_u: unconfined_r: unconfined_t: s0-s0: c0.c1023
```

Tale account è mappato per attuare un comportamento di default e quindi potrà eseguire processi o applicativi come utente di tipo unconfined_u e avere accesso a file, directory o processi etichettati con il ruolo di tipo unconfined_r. Tuttavia, questo account non avrà però, il diritto di avviare alcun processo all'interno del sistema. Il seguente blocco di codice mostra che *restricteduser* sta tentando di avviare il *demone httpd*, ma verrà restituito un errore.

```
[restricteduser@localhost ~]$ service httpd start

#output
Redirecting to /bin/systemctl start httpd.service
Failed to issue method call: Access denied
```

A questo punto, aggiungerò l'account restricteduser al file /etc/sudoers, andando ad aggiungere la riga *restricteduser ALL = (ALL) ALL*. Questa azione consentirà a tale account di utilizzare i privilegi di root.

Dopo questa fase di modifica del file, allo scopo di assegnare i permessi di root all'account *restricteduser*, proverò a ripetere il processo di avvio del *demone httpd* e tutto andrà a buon fine.

```
[restricteduser@localhost ~]$ sudo service httpd start

#output
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for restricteduser:
Redirecting to /bin/systemctl start httpd.service
```

Allo stesso modo sarà anche possibile sospendere tale servizio. A volte potrebbe però essere necessario, impedire ad un particolare utente (nel mio caso ho preso come esempio l'account restricteduser) di avviare un servizio (come ad esempio httpd) anche quando l'account dell'utente è elencato nel file sudoers.

Questo sarà possibile mappando tale account all'utente di tipo user_u, con il comando seguente:

```
login semanage -a -s user_u restricteduser
```

Ora che restricteduser è stato limitato a user_u (questo implica che tale account verrà limitato anche al ruolo user_r e dominio user_t), verificherò l'accesso utilizzando il comando indicato di seguito:

```
seinfo -uuser_u -x

#output:
   user_u
      default level: s0
      range: s0
      roles:
         object_r
         user_r
```

Con il comando successivo, verificherò se il ruolo user_r è autorizzato per il dominio httpd_t.

```
seinfo -ruser_r -x | grep httpd


#output:
        httpd_user_htaccess_t
        httpd_user_script_exec_t
        httpd_user_ra_content_t
        httpd_user_rw_content_t
        httpd_user_script_t
        httpd_user_content_t
```

L'output di cui sopra mostra che non è presente il dominio httpd_t. Questo implica che con l'account restricteduser, non sarà possibile avviare il *demone httpd*, poichè l'accesso viene negato perché il processo httpd viene eseguito all'interno del dominio httpd_t e questo, non è uno dei domini a cui il ruolo user_r, assunto dal tipo di utente user_u è autorizzato ad accedere.

```
[restricteduser@localhost ~]$ sudo service httpd start

#output:
sudo: PERM_SUDOERS: setresuid (-1, 1, -1): Operation not permitted
```

Fino a questo punto ho descritto i primi tre attributi che permettono di definire i contesti di sicurezza per processi, utenti o risorse, ovvero l'utente SELinux, il ruolo SELinux ed il tipo o dominio SELinux. Il quarto campo del contesto di sicurezza mostra la sensibilità e facoltativamente, la categoria della risorsa. Questo attributo definisce una sicurezza di tipo multilivello e per capire meglio tale approccio, considererò il contesto di sicurezza del file di configurazione del damone FTP.

```
ls -Z /etc/vsftpd/vsftpd.conf

#output:
-rw -------. root root system_u: object_r: etc_t:
s0 /etc/vsftpd/vsftpd.conf
```

Il quarto campo del contesto di sicurezza mostra una sensibilità di s0. La sensibilità è parte del meccanismo di sicurezza multi-livello di tipo gerarchico. Infatti, per gerarchia si vuole intendere che i livelli di sensibilità possono andare sempre più in profondità per un contenuto più sicuro nel file system. Il livello 0 (rappresentato da s0) è il livello di sensibilità più basso, paragonabile a "pubblico" . Possono esserci altri livelli di sensibilità con valori di s più alti, come ad esempio internal, confidential, o regulatory, i quali possono essere rappresentati rispettivamente da s1, s2 e s3. Questa mappatura non è prevista dalla policy, infatti gli amministratori di sistema possono configurare cosa significhi ciascun livello di sensibilità.

Quando un sistema abilitato SELinux utilizza MLS per il suo tipo di policy (configurato nel file /etc/selinux/config), può contrassegnare determinati file e processi con determinati livelli di sensibilità. Il livello più basso è chiamato "current sensitivity" , mentre il livello più alto è prende il nome di "clearance sensitivity" .

Le categorie (identificate dalla lettera c, possono essere considerate come etichette assegnate ad una specifica risorsa. Esempi di categorie possono essere nomi di reparto, nomi di clienti o anche progetti. Lo scopo della categorizzazione è quindi quello di perfezionare ulteriormente il controllo degli accessi. Ad esempio, è possibile contrassegnare determinati file con sensibilità riservata per gli utenti da due diversi reparti interni.

Per i contesti di sicurezza SELinux, la sensibilità e la categoria funzionano insieme quando viene implementata una categoria. Quando si utilizza un intervallo di livelli di sensibilità, il formato mostra i livelli di sensibilità separati da un trattino come ad esempio s0-s2. Quando invece si utilizza una categoria, viene visualizzato un intervallo con un punto in mezzo. I valori di sensibilità e categoria sono invece separati dai due punti.

```
user_u: object_r: etc_t: S0: c0-c2
```

In questo caso è presente un solo livello di sensibilità, il quale è identificato con s0. Il livello di categoria è invece compreso tra c0 e c2. Ad esempio, un processo avrà l'accesso in lettura ad una determinata risorsa solo quando la sua sensibilità e il livello di categoria risulterà superiore a quello della risorsa. Ovvero, quando il dominio del processo domina il tipo di risorsa. Il processo potrà invece scrivere sulla risorsa quando il livello di sensibilità/categoria risulterà essere inferiore a quello della risorsa stessa.

4.4.1 SELinux in Docker, Rkt, LXC e LXD

L'interazione tra la policy di SELinux e Docker si concentra su due aspetti di sicurezza, i quali si basano sulla protezione dell'host e la protezione dei container l'uno dall'altro. Come detto nella sezione precedente, le etichette SELinux sono composte da quattro parti:

```
User:Role:Type:level.
```

SELinux in Docker può essere utilizzato in modalità *Type Enforcement* e *MCS MLS*.

Nel caso del *Type Enforcement*, come descritto nella sezione precedente, l'imposizione dei tipi è un tipo di imposizione in cui le regole sono basate sul tipo di processo. Il tipo predefinito per un processo all'interno di un container è identificato e limitato a svirt_lxc_net_t. Questo tipo è autorizzato a leggere ed eseguire tutti i "tipi" di file sotto la directory /usr e una grande parte dei tipi sotto la directory /etc. svirt_lxc_net_t è invece autorizzato ad usare la rete ma senza il permesso di leggere il contenuto all'interno delle directory /var, /home, /root, e /mnt. Svirt_lxc_net_t è inoltre autorizzato a scrivere solo sui file etichettati come svirt_sandbox_file_t e come docker_var_lib_t. Tutti i file in uno specifico container saranno invece etichettati come svirt_sandbox_file_t. L'accesso a docker_var_lib_t è consentito allo scopo di poter consentire l'utilizzo dei Docker Volumes [50].

La separazione MCS in Docker come in Rkt, il quale descriverò nella sezione successiva, prende anche il nome di *Svirt* e funziona assegnando un valore univoco al campo "livello" dell'etichetta SELinux di ciascun container. Per impostazione di default, a ciascun container viene assegnato il livello MCS equivalente al PID del processo che avvia il container in questione. Questo campo può essere utilizzato anche negli ambienti MLS in cui viene impostato il campo relativo alla sicurezza multilivello ad esempio a "TopSecret" o "Secret" .

La policy di default, include regole che impongono che le etichette MCS del processo debbano necessariamente "dominare" l'etichetta MCS del target. Come target si può intendere ad esempio un file per il quale si vogliono controllare gli accessi in lettura e/o scrittura. L'etichetta MCS come anticipato nella sezione di qui sopra, somiglia a qualcosa di questo tipo: s0: c1, c2. La definizione di questo tipo di etichetta, avrà lo scopo di "dominare" i file etichettati s0, s0: c1, s0: c2, s0: c1, c2. Tuttavia, non sarebbe abilitata a "dominare" file etichettati come s0: c1, c3. Tutte le etichette MCS sono quindi definite in modo da poter utilizzare due categorie. Questo

garantisce che non ci siano due container che possono avere la stessa etichetta MCS impostata di default. I file con s0 i quali sono identificati nella maggior parte dei file presenti sul sistema, non sono bloccati a livello di sicurezza MCS. Infatti, l'accesso a tali file, viene regolato salla base del Type Enforcement e sarebbe quindi buona norma, applicare il MCS.
Rkt invece, fornisce il supporto a SELinux per la sicurezza dei container, utilizzando *Svirt*. *Svirt* è una tecnologia che è stata inclusa in Red Hat Enterprise Linux 6, la quale integra SELinux e la virtualizzazione. Svirt ha lo scopo di applicare la sicurezza a livello MAC, per migliorarne la sicurezza stessa nei caso di utilizzo delle macchine virtuali. I motivi principali per l'integrazione di queste tecnologie consistono nel voler migliorare la sicurezza e l'indebolimento del sistema contro i eventuali bug nell'hypervisor, che potrebbero essere utilizzati come vettore di attacco diretto verso l'host o verso un'altra macchina virtuale. Quando ad esempio si prendono in considerazione due differenti servizi che non sono stati virtualizzati, si considera anche che le macchine sulla quelle questi sono in esecuzione siano fisicamente separate. Quindi, qualsiasi exploit viene solitamente contenuto nella macchina interessata, facendo eccezione agli attacchi di rete. Quando invece i servizi sono raggruppati in un ambiente di tipo virtualizzato, nel sistema emergono ulteriori vulnerabilità. Infatti, la presenza di un difetto di sicurezza nell'hypervisor potrebbe essere sfruttato da un'istanza all'inerno di uno specifico container, questo ospite potrà quindi essere in grado non solo di attaccare l'host, ma anche altri container che stanno eseguendo applicazioni sempre in quello specifico host. Questo tipo di attacchi possono estendersi fino ad esporre altri i container in esecuzione da parte di altri utenti a tale attacco. Svirt ha quindi lo scopo di gestire i container e di limitare la loro capacità di lanciare ulteriori attacchi. SELinux con Svirt, introduce quindi un framework di sicurezza collegabile per istanze virtualizzate nella sua implementazione di tipo *Mandatory Access Control*. Il framework Svirt consente ai guest users e quindi i loro container e le loro risorse, di essere etichettati in modo univoco. Una volta etichettati, sarà possibile applicare le policy di sicurezza SELinux, allo scopo di poter permettere o rifiutare l'accesso tra i diversi container.

All'avvio, rkt tenterà di leggere /etc/selinux/(policy)/contexts/ lxc_contexts. Se questo file non esiste, non verranno eseguite transizioni SELinux. Se lo fa, rkt genererà un contesto per istanza. Tutti i supporti per l'istanza verranno creati utilizzando il contesto file definito in lxc_contexts e i processi di istanza verranno eseguiti in un contesto derivato dal contesto del processo definito in lxc_contexts.

I processi avviati in questi contesti non saranno in grado di interagire con processi o file in qualsiasi contesto di un'altra istanza, anche se sono in esecuzione come lo stesso utente. Le singole distribuzioni Linux possono imporre ulteriori vincoli di isolamento in questi contesti. Per ulteriori dettagli, consultare la documentazione di distribuzione.

LXC, a partire dalla versione 2.0 ed LXD utilizzano invece il *Linux Security Module* AppArmor di default.

4.5 User Namespace

Come descritto nella sezione 3.1.1, il principio dei namespaces 3.1.1 si basa sulla definizione di un'etichetta (namespace), che viene associata ad un gruppo di risorse di sistema e di seguito ne descriverò alcuni esempi pratici sul funzionamento relativo alle varie tecnologie di gestione dei container.

4.5.1 User Namespace in Docker

Prima di effettuare una dimostrazione pratica relativa all'utilizzo dei namespaces 3.1.1, verrà descritto il problema relativo a ciò che un utente root può fare se l'user namespaces non è abilitato. Uno delle funzionalità concesse da Docker è che l'utente può avere privilegi di root sui container in modo tale che l'utente stesso possa installare facilmente i pacchetti o applicativi software. Ma questo è come una spada a doppio taglio nella tecnologia del container Linux. Infatti, con un pò di torsione, un utente non-root può accedere come root nel sistema host, come ad esempio in /etc.

```
# Avvio un container ed eseguo il mount di host1 /etc onto /root/etc
$ docker run --rm -v /etc:/root/etc -it ubuntu

# Effettuo alcuni cambiamenti all'interno di /root/etc/hosts
root@37ef67889325:/# vi /root/etc/hosts

# Esco dal container
root@37ef67889325:/# exit

# Controllo /etc/hosts
$ cat /etc/hosts
```

Nei passaggi di cui sopra, ho mostrato come sia semplice poter acquisire i permessi di root sul sistema host da parte di un utente non-root.
Di seguito descriverò un esempio di come sia possibile ovviare a questo problema utilizzando gli user namespaces.

```
# Creo uno user chiamato "user1"
$ sudo adduser user1

# Effettuo il setup subuid and subgid, impostando il relativi
Setuid (set id utente) è un bit di autorizzazione che consente
agli utenti di eseguire un programma con le autorizzazioni del
proprietario.
Setgid (id del gruppo) invece, è un bit che se
settato ad 1 consente all'utente di eseguire un programma
con le autorizzazioni del proprietario del gruppo.
$ sudo sh -c 'echo user1:500000:65536 > /etc/subuid'
$ sudo sh -c 'echo user1:500000:65536 > /etc/subgid'
```

In seguito effettuerò una modifica nel file /etc/init.d/docker, aggiungendo dopo /usr/local/bin/-docker daemon –userns-remap=default ed in seguito eseguo il riavvio di Docker.
A questo punto si andrà a verificare che user namespaces funzioni in modo corretto.

```
# Avvio un container ed eseguo il mount di host1 /etc onto /root/etc
$ docker run --rm -v /etc:/root/etc -it ubuntu

# Controllo gli owner dei file in /root/etc, e dovrei visualizzare
"nobody␣nogroup".
root@d5802c5e670a:/# ls -la /root/etc
total 180
drwxr-xr-x 11 nobody nogroup 1100 Mar 21 23:31 .
drwx------ 3 root root 4096 Mar 21 23:50 ..
lrwxrwxrwx 1 nobody nogroup 19 Mar 21 23:07 ..
-rw-r--r-- 1 nobody nogroup 48 Mar 10 22:09 boot2docker
drwxr-xr-x 2 nobody nogroup 60 Mar 21 23:07 default
 :
 :

# Provo a creare un file in /root/etc
root@d5802c5e670a:/# touch /root/etc/test
touch: cannot touch '/root/etc/test': Permission denied

# Provo a cancellare un file in /root/etc
root@d5802c5e670a:/# rm /root/etc/hostname
rm: cannot remove '/root/etc/hostname': Permission denied
```

Ora ogni tentativo di creazione o rimozione di un file nella cartella root mi è stato negato.

4.5.2 User Namespace in Rkt

Allo stesso modo è possibile modificare gruppo utente e identificativo utente anche con Rkt, utilizzando i flags –user e –group al momento dell'avvio di un'immagine all'interno di un container Rkt. Di default GID e UID son specificati nell' ACI manifest che è un file JSON che include i dettagli dell'immagine e le informazioni relative alla sua esecuzione.

```
# rkt --insecure-options=image run docker://busybox --user=1000
 --group=100 --exec id
```

4.5.3 User Namespace in LXC

LXC nelle versioni precedenti alla 1.0 veniva cosiderato insicuro dal lato dell' isolamento, poichè mentre l'esecuzione di un processo veniva fatta in uno spazio dei nomi separato, il valore di uid0 all'interno del container, rimaneva lo stesso del valore all'esterno del container, il che significava che se si otteneva in un qualche modo l'accesso a qualsiasi risorsa host attraverso proc, sys o alcune syscall casuali, si acquisiva la possibilità di uscire dal container acquisendo inoltre i permessi di root sull' intero sistema host.

Invece, dalla versione di LXC 1.0 risulta possibile avviare un Full system container, interamente come user e non come root.

Ad ogni utente autorizzato all'utilizzo dei namespaces, gli vengono assegnati degli UID e GID inutilizzati, che idealmente sono degli interi non superiori a 65536. UID e GID possono quindi essere utilizzati tramite due strumenti che sono newuidmap e newgidmap, i quali consentono di mappare gli UID e GID a UID/GID virtuali in un namespace utente.

Diventa quindi possibile creare un container utilizzando la seguente configurazione:

```
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

Il che significa che avrò una mappatura di UID e GID definita per il mio container e che gli UID e GID da 0 a 65536 nel container saranno mappati a UID/GID da 100000 a 165536 nell'host.

Per farè ciò avrò però bisogno di avere gli intervalli assegnati al mio utente a livello di sistema con i seguenti comandi:

```
alias@host:~$ grep alias /etc/sub* 2>/dev/null
/etc/subgid:alias:100000:65536
/etc/subuid:alias:100000:65536
```

A questio punto, LXC è stato aggiornato in modo che tutti gli strumenti siano consapevoli che i container con GID/UID in quel range non sono privati. Inoltre, i percorsi standard avranno anche gli equivalenti non privilegiati:

```
/etc/lxc/lxc.conf => ~ / .config / lxc / lxc.conf
/etc/lxc/default.conf => ~ / .config / lxc / default.conf
/ var / lib / lxc => ~ / .local / share / lxc
/ var / lib / lxcsnaps => ~ / .local / share / lxcsnaps
/ var / cache / lxc => ~ / .cache / lxc
```

4.5.4 User Namespace in LXD

LXD come impostazione di default invece, utilizza i container senza alcun privilegio di root. Ovvero la differenza tra un container con privilegi di root ed uno senza privilegi, sta nel fatto che l'utente principale che sta eseguendo un container sia o meno lo stesso utente principale nel sistema host e che sia identificato a livello kernel come uid0.

Il modo in cui vengono creati i container non privilegiati è simile ad LXC, e si basa sul prendere un insieme di UID e GID normali dall'host, di solito almeno 65536 UID e lo stesso per i GID, questo per essere conformi a POSIX e poterli mappare nel container.

Quindi si avrà come impostazione predefinita una mappa di 65536 UID e GID, con un ID di base host di 100000. Questo implica che root nel container (uid 0) verrà mappato all'host con uid 100000 e l'uid 65535 all'inerno del container, sarà mappato a uid 165535 sull'host. Valori di UID/GID uguali a 65536 o superiori, non vengono quindi mappati e restituiranno un errore nel caso in cui si tenti di utilizzarli.

Dal punto di vista della sicurezza, questo significa che quello che non è permesso agli utenti e gruppi mappati all'interno del container, non sarà accessibile. Quindi tale risorsa definita come inaccessibile, apparirà come proprietà di uid / gid "-1" (o nobody/nogroup in userspace). Questo implica che anche se ci fosse un modo per uscire al di fuori del container, anche se si era root all'interno di esso, all'esterno ci si troverebbe con altrettanti privilegi sull'host ma settati come nobody/nogroup user.

LXD inoltre offre una serie di opzioni relative alla configurazione non privilegiata come:

- Aumentare la dimensione della mappa predefinita UID/GID
- Impostazione di mappe per container
- Inserimento di "buchi" nella mappa per esporre gli utenti e i gruppi di host

Come ho accennato in precedenza, LXD di base ha una mappa di default composta da 65536 UID/GID, dove nella maggior parte dei casi non sarà necessario modificarla. Ci sono tuttavia alcuni casi in cui è possibile modificarla, come ad esempio quando si ha di accesso a UID/GID superiori a 65535. Questo risulta essere più comune quando si utilizza l'autenticazione di rete all'interno dei propri container.

Un altro caso si ha quando si desidera utilizzare mappe per ciascun container. In questo caso infatti, si avrà bisogno di 65536 UID/GID disponibili per container.

Oppure quando si vuole puntare ad alcuni "buchi" nella mappa di un determinato container perchè si ha bisogno di accedere all'host o ai GID dell'host.

La mappa predefinita è di solito controllata dall'insieme di utilità e file di "shadow" . Sui sistemi in cui è necessario, i file "/ etc / subuid" e "/etc/subgid" vengono utilizzati per configurare queste mappe.
Di seguito descriverò un esempio di configurazione.

```
alias@host1:~$ cat /etc/subuid
lxd:100000:65536
root:100000:65536

alias@host1:~$ cat /etc/subgid
lxd:100000:65536
root:100000:65536
```

E' importante che le mappe per "lxd" e "root" siano sempre mantenute in sincronia. LXD è limitato dall'allocazione "root" . La voce "lxd" invece, viene utilizzata per tenere traccia di ciò che è necessario rimuovere se LXD viene disinstallato.

Nel caso in cui sia necessario aumentare la dimensione della mappa disponibile per LXD, basta modificare entrambi gli ultimi valori da 65536 a qualsiasi dimensione risulti a noi necessaria.

```
alias@host1:~$ cat /etc/subuid
lxd:100000:1000000000
root:100000:1000000000

alias@host1:~$ cat /etc/subgid
lxd:100000:1000000000
root:100000:100000000
```

Ho impostato un valore così alto in modo da non dovermi preoccupare in futuro.

Dopo questi passaggi sarà necessario riavviare LXD, poichè la nuova mappa venga rilevata.

```
root@host1:~# systemctl restart lxd
root@host1:~# cat /var/log/lxd/lxd.log
lvl=info msg="LXD␣2.14␣is␣starting␣in␣normal␣mode"
path=/var/lib/lxd t=2017-10-14T21:21:13+0000
lvl=warn msg="CGroup␣memory␣swap␣accounting␣is␣disabled,
␣swap␣limits␣will␣be␣ignored." t=2017-10-14T21:21:13+0000
lvl=info msg="Kernel␣uid/gid␣map:" t=2017-10-14T21:21:13+0000
lvl=info msg="␣-␣u␣0␣0␣4294967295" t=2017-10-14T21:21:13+0000
lvl=info msg="␣-␣g␣0␣0␣4294967295" t=2017-10-14T21:21:13+0000
lvl=info msg="Configured␣LXD␣uid/gid␣map:" t=2017-10-14T21:21:13+0000
lvl=info msg="␣-␣u␣0␣1000000␣1000000000" t=2017-10-14T21:21:13+0000
lvl=info msg="␣-␣g␣0␣1000000␣1000000000" t=2017-10-14T21:21:13+0000
lvl=info msg="Connecting␣to␣a␣remote␣simplestreams␣server"
 t=2017-10-14T21:21:13+0000
lvl=info msg="Expiring␣log␣files" t=2017-10-14T21:21:13+0000
lvl=info msg="Done␣expiring␣log␣files" t=2017-10-14T21:21:13+0000
lvl=info msg="Starting␣/dev/lxd␣handler" t=2017-10-14T21:21:13+0000
lvl=info msg="LXD␣is␣socket␣activated" t=2017-10-14T21:21:13+0000
lvl=info msg="REST␣API␣daemon:" t=2017-10-14T21:21:13+0000
lvl=info msg="␣-␣binding␣Unix␣socket" socket=/var/lib/lxd/
unix.socket t=2017-10-14T21:21:13+0000
lvl=info msg="␣-␣binding␣TCP␣socket" socket=[::]:
8443t=2017-06-14T21:21:13+0000
lvl=info msg="Pruning␣expired␣images" t=2017-10-14T21:21:13+0000
lvl=info msg="Updating␣images" t=2017-10-14T21:21:13+0000
lvl=info msg="Done␣pruning␣expired␣images" t=2017-10-14T21:
21:13+0000
lvl=info msg="Done␣updating␣images" t=2017-10-14T21:21:13+0000
root@host1:~#
```

A questo punto, la mappa configurata viene registrata all'avvio LXD e può essere utilizzata per confermare che la riconfigurazione ha funzionato come previsto.

Il passaggio successivo, sarà quello di riavviare tutti i container affinché possano iniziare a utilizzare la nuova mappa espansa.

A condizione che si abbia una quantità sufficiente di UID/GID assegnabili a LXD, è possibile configurare i propri container per utilizzare un'allocazione non sovrapposta di UID e GID.

Questo può essere utile per due motivi, ovvero quando si sta eseguendo un software che modifica ulimit delle risorse del kernel, o quando questi limiti specificati per l'utente sono legati a un kernel UID e dovranno attraversare i confini dei container, i quali portano a problemi difficili da eseguire in debug, quando, nel caso in cui un container può eseguire un'azione, ma tutti gli altri container non sono in grado di fare altrettanto. Quindi per poter disporre di un container utilizzando la propria mappa distinta, basterà eseguire i comandi elencati di seguito.

```
root@host1:~$ lxc config set test security.idmap.isolated true
roothost1:~$ lxc restart test
root@host1:~$ lxc config get test volatile.last_state.idmap
```

```
[{"Isuid":true,"Isgid":false,"Hostid":165536,"Nsid":0,
"Maprange":65536},{"Isuid":false,"Isgid":true,
"Hostid":165536,"Nsid":0,"Maprange":65536}]
```

Il passaggio di riavvio è necessario perchè LXD rimappi l'intero filesystem del container nella nuova mappa.

Questo passaggio richiederà dunque una quantità variabile di tempo a seconda del numero di file nel container e della velocità della memoria.

Quindi, dopo il riavvio, il container avrà la propria mappa di 65536 UID/GID.

Se invece si volessero allocare più di 65536 UID/GID predefiniti in un container isolato, questo sarà possibile con i comandi descritti di seguito.

```
alias@host1:~$ lxc config set test security.idmap.size 200000
alias@host1:~$ lxc restart test
alias@host1:~$ lxc config get test volatile.last_state.idmap
[{"Isuid":true,"Isgid":false,"Hostid":165536,"Nsid":0,
"Maprange":200000},{"Isuid":false,"Isgid":true,"Hostid":165536,
"Nsid":0,"Maprange":200000}]
```

Nel caso in cui si stesse cercando di assegnare più UID o GID di quanti siano stati realmente resi disponibili, LXD mostrerà il messaggio indicato di seguito.

```
alias@host1:~$ lxc config set test security.idmap.size
2000000000
error: Not enough uid/gid available for the container.
```

Il fatto che tutte le UID e GID in un container non privilegiato siano mappate ad un intervallo normalmente inutilizzato nell'host significa che la condivisione di dati tra host e container risulta a tutti gli effetti impossibile.

Quindi se fosse necessario condividere la propria home directory dell'utente con un container specifico, bisognerebbe definire una nuova voce "disk" in LXD che passa la home directory al container in questione.

```
alias@host1:~$ lxc config device add test home disk source=
/home/alias path=/home/ubuntu
Device home added to test
```

Questo però non basta, poichè anche se il supporto risulta essere chiaramente lì, è completamente inaccessibile al container.

```
alias@host1:~$ lxc exec test -- bash
root@test:~# ls -lh /home/
total 529K
drwx--x--x 45 nobody nogroup 84 Jun 14 20:06 ubuntu
```

Per risolvere questo problema, è quindi necessario fare alcuni passi aggiuntivi come consentire ad LXD di utilizzare il nostro UID e GID, riavviare LXD in modo da poter caricare la nuova mappa, impostare una mappa personalizzata per il nostro container e riavviare il container per applicare la nuova mappa.

```
alais@host1:~$ printf "lxd:$(id␣-u):1\nroot:$(id␣-u):1\n" |
sudo tee -a /etc/subuid
lxd:201105:1
root:201105:1

alias@host1:~$ printf "lxd:$(id␣-g):1\nroot:$(id␣-g):1\n" |
sudo tee -a /etc/subgid
lxd:200512:1
```

```
root:200512:1

alias@host1:~$ sudo systemctl restart lxd

alias@host1:~$ printf "uid␣$(id␣-u)␣1000\ngid␣$(id␣-g)␣1000" |
 lxc config set test raw.idmap -

alias@host1:~$ lxc restart test
```

A quel punto, il container funzionerà nel modo corretto.

```
alias@host1:~$ lxc exec test -- su ubuntu -l
ubuntu@test:~$ ls -lh
total 119K
drwxr-xr-x 5 ubuntu ubuntu 8 Nov 18 2017 data
drwxr-x--- 4 ubuntu ubuntu 6 Nov 13 17:05 Desktop
drwxr-xr-x 3 ubuntu ubuntu 28 Nov 13 20:09 Downloads
drwx------ 84 ubuntu ubuntu 84 14 2017 Maildir
drwxr-xr-x 4 ubuntu ubuntu 4 Nov 20 15:38 snap
ubuntu@test:~$
```

4.6 Bridge Networking

Di seguito descriverò le varie funzionalità fornite dai container management systems per la creazione, gestione e isolamento delle reti e delle loro interfacce di rete all'interno dei container, rispetto alle reti esterne. Per maggiori dettagli fare riferimento alla documentazione Docker dalla quale mi sono ispirato [42]

4.6.1 Bridge Networking in Docker

Docker, all'atto della sua installazione crea automaticamente tre reti. Tali reti è possibile visualizzarle utilizzando il comando network ls di docker.

```
$ docker network ls

NETWORK ID NAME DRIVER
7fca4eb8c647 bridge bridge
9f904ee27bf5 none null
cf03ee007fb4 host host
```

Queste tre reti sono incorporate in Docker. Quando si esegue un container, è possibile utilizzare il comando –network per specificare le reti connesse al container.
La rete bridge rappresenta la rete docker0 che è presente di default in tutte le installazioni Docker, a meno che non ne venga specificata una differente con l'opzione –network = (NETWORK). Come impostazione di default, il Docker-daemon si occuperà di collegare i container a tale rete. È possibile inoltre, vedere questo bridge come parte dello stack di rete di un host utilizzando il comando ip addr show sull'host.

```
$ ip addr show

docker0 Link encap:Ethernet HWaddr 02:42:47:bc:3a:eb
          inet addr:172.17.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
          inet6 addr: fe80::42:47ff:febc:3aeb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:9001 Metric:1
          RX packets:17 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1100 (1.1 KB) TX bytes:648 (648.0 B)
```

La rete "none" invece, aggiunge un container a uno stack di rete specifico per il container. Questo container però, non avrà alcuna interfaccia di rete.

La rete host invece, si occupa di aggiungere un container nello stack di rete dell'host. Quindi per quanto riguarda la rete, non esiste alcun isolamento tra la macchina host e il container stesso. Ad esempio, se si mandasse in esecuzione un container che esegue un server Web sulla porta 80 utilizzando la rete host, il server web rimarrà in ascolto sulla porta 80 della macchina host, anche se è stato avviato all'interno di un container.

Le reti none e host non sono direttamente configurabili in Docker, ma risulta comunque possibile configurare la rete bridge di default, nonché le reti bridge definite dall'utente stesso.
Come detto in precedenza, la rete bridge di default è presente su tutti gli host Docker e se non viene specifica una rete differente, i nuovi container verranno automaticamente agganciati alla rete bridge di default.
Con il comando docker network inspect possiamo vedere le informazioni relative ad una specifica rete:

```
$ docker network inspect bridge

[
   {
       "Name": "bridge",
       "Id": "f7ab26d71........",
       "Scope": "local",
       "Driver": "bridge",
       "IPAM": {
           "Driver": "default",
           "Config": [
               {
                   "Subnet": "172.17.0.1/16",
                   "Gateway": "172.17.0.1"
               }
           ]
       },
       "Container": {},
       "Options": {
           "com.docker.network.bridge.default_bridge": "true",
           "com.docker.network.bridge.enable_icc": "true",
           "com.docker.network.bridge.enable_ip_masquerade": "true",
           "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
           "com.docker.network.bridge.name": "docker0",
           "com.docker.network.driver.mtu": "9001"
       },
       "Labels": {}
   }
]
```

Di seguito descriverò un esempio di due container collegati ciascuno alla rete bridge di default.

```
$ docker run -itd --name=container1 busybox

3386a527aa08b37ea92.....

$ docker run -itd --name=container2 busybox

94447ca479852d29aed.....
```

Ispezionando nuovamente la rete bridge, sarà possibile vedere che entrambi i container ora sono connessi alla rete bridge.

```
$ docker network inspect bridge

{[
    {
        "Name": "bridge",
        "Id": "f7ab26d71dbd.....",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.1/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Container": {
            "3386a527aa0......": {
                "EndpointID": "647c12443.....",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "94447ca4798.....": {
                "EndpointID": "b047d090f.....",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "9001"
        },
        "Labels": {}
    }
]
```

I container collegati alla rete bridge di default possono quindi comunicare tra di loro tramite l'indirizzo IP. Docker però non supporta la ricerca automatica dei servizi sulla rete bridge di default. Infatti se si desidera che i container siano in grado di risolvere gli indirizzi IP del container per nome, è necessario utilizzare delle reti definite dall'utente. È inoltre possibile collegare due container assieme, utilizzando l'opzione legacy dock run –link, ma a meno che non sia strettamente necessario, non è opportuno collegare più container tra di loro.
È inoltre possibile, tramite con il comando attach, vedere come appare la rete dall'interno di un container.

```
$ docker attach container1

root@3386a527aa08:/# ip -4 addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    inet 127.0.0.1/8 scope host lo
```

```
       valid_lft forever preferred_lft forever
633: eth0@if634: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>
mtu 1500 qdisc noqueue
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
```

Dall'interno del container utilizzerò il comando ping per verificare l'effettiva connessione tra i 2 container.

```
root@3386a527aa08:/# ping -w3 172.17.0.3

PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.096 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.080 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.074 ms

--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.074/0.083/0.096 ms
```

Utilizzando il comando cat invece, sarà possibile visualizzare il file / etc / hosts sul container. Questo è utile, perchè ci mostra gli hostname e indirizzi IP che vengono riconosciuti dal container.

```
root@3386a527aa08:/# cat /etc/hosts

172.17.0.2 3386a527aa08
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

La rete bridge di default docker0 supporta inoltre l'utilizzo della mappatura delle porte e docker run –link, al fine di poter consentire lecomunicazioni tra i container nella rete docker0. Ma tale approccio sarebbe da evitare, utilizzando invece gli user-defined network bridge.

Quindi se avessimo la necessità di far comunicare più container tra di loro, sarebbe una buona prassi utilizzare le user-defined network bridge, oltre ad abilitare la risoluzione automatica dei nomi dei container e dei loro indirizzi ip. Docker fornisce i driver per la creazione di queste reti, infatti è possibile creare oltre alle reti bridge anche delle OVERLAY network o MACVLAN network. Inoltre si possono creare anche plugin di rete o reti remote.
Docker quindi fornisce la possibilità di creare tante reti quante realmenente ne abbiamo bisogno, pe poi collegare in qualsiasi momento uno specifico container ad una o più di queste reti. Inoltre, è possibile collegare e disconnettere i container in esecuzione dalle reti senza dover necessariamente riavviare il container in questione. Quando un container è collegato a più reti, la sua connettività esterna viene fornita tramite la prima rete non interna, seguendo un ordine lessicale.

Di seguito descriverò nel dettaglio i driver di rete incorporati in Docker.
Una rete bridge, è il tipo più comune di rete utilizzata in Docker. Tali reti sono simili a quelle di default, ma aggiungono alcune nuove funzionalità e ne rimuovono di vecchie. Di seguito descriverò alcuni esempi di reti bridge sui container.

```
$ docker network create --driver bridge isolated_nw

1196a4c5af4......

$ docker network inspect isolated_nw
```

```
[
    {
        "Name": "isolated_nw",
        "Id": "1196a4c5af......",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1/16"
                }
            ]
        },
        "Container": {},
        "Options": {},
        "Labels": {}
    }
]

$ docker network ls

NETWORK ID NAME DRIVER
9f904ee27bf5 none null
cf03ee007fb4 host host
7fca4eb8c647 bridge bridge
c5ee82f76de3 isolated_nw bridge
```

Dopo aver creato la rete, è possibile lanciare i container associandoli ad essa utilizzando l'opzione –network = (NETWORK) di docker.

```
$ docker run --network=isolated_nw -itd --name=container3 busybox

8c1a0a5be48......

$ docker network inspect isolated_nw
[
    {
        "Name": "isolated_nw",
        "Id": "1196a4c5af43a21ae38ef34515b6af19
␣␣␣␣␣␣␣␣236a3fc48122cf585e3f3054d509679b",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {}
            ]
        },
        "Container": {
"8c1a0a5be4......": {
                "EndpointID": "93b2db4a.......",
                "MacAddress": "02:42:ac:15:00:02",
                "IPv4Address": "172.21.0.2/16",
                "IPv6Address": ""
            }
        },
```

```
        "Options": {},
        "Labels": {}
    }
]
```

Ovviamente i container da lanciare in questa rete devono risiedere nello stesso host di Docker, inoltre ogni container della rete può comunicare immediatamente con gli altri container della rete, anche se la rete isola i container dalle reti esterne. Se all'interno della rete bridge si volesse che

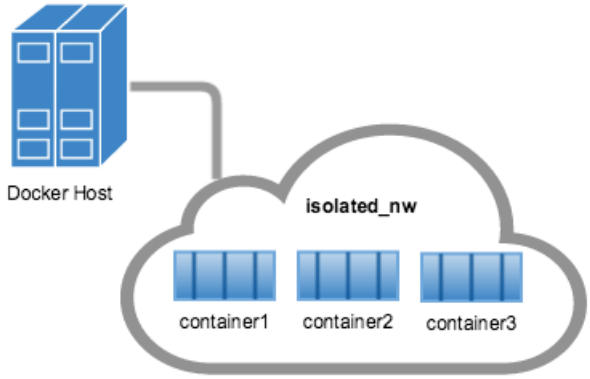

Figura 4.4. Bridged network [42].

una porzione di essa sia disponibile alle reti esterna, sarà necessario esporre e pubblicare le porte dei container all'interno di essa.

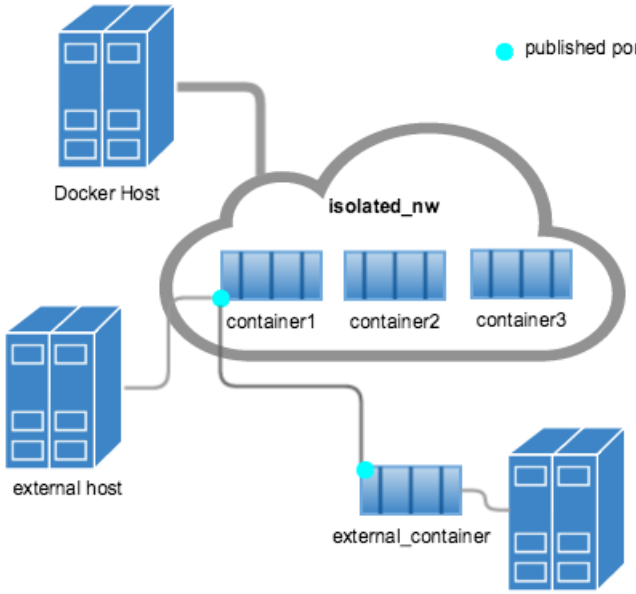

Figura 4.5. Exposed bridged network [42].

4.6.2 Bridge Networking in LXC

Utilizzando LXC Linux container invece vi sono tipicamente due modi per poter creare un'installazione di rete per un container, ovvero host-sared bridge e indipendent bridge.

Host-shared bridge crea un bridge dall'interfaccia di rete principale, il quale conterrà sia l'IP dell'host che gli indirizzi IP del container in questione.

Indipendent bridge invece, crea un network bridge differente, collegando tra loro e con il bridge stesso i container in esecuzione sull'host, ma nel caso in cui ciascun container volesse uscire su internet e non comunicare in locale, si andrà ad utilizzare il port forwarding.

Quindi con la prima tipologia di bridge, sarà possibile acquisire una locazione tramite richiesta al server DHCP dalla rete host. Il secondo tipo invece è mascherato all'interno della propria rete interna, ovvero si trovs dietro ad un NAT firewall e riceve solo attraverso le porte a lui inoltrate.

lxc-net utilizza il secondo tipo, ovvero una configurazione basata su lxc-net è una configurazione mascherata.

Tipicamente entrambi i tipi di configurazioni di bridge networking utilizzano l'impostazione della rete veth per ciascun container.

Dalla versione di LXC 2.0 la seconda tipologia di bridge networking può essere utilizzata attraverso la componente lxc-net.

Di seguito ho riportato una configurazione di esempio per lxc-net.

```
USE_LXC_BRIDGE="true"
LXC_BRIDGE="lxcbr0"
LXC_ADDR="10.0.3.1"
LXC_NETMASK="255.255.255.0"
LXC_NETWORK="10.0.3.0/24"
LXC_DHCP_RANGE="10.0.3.2,10.0.3.254"
LXC_DHCP_MAX="253"
LXC_DHCP_CONFILE=""
LXC_DOMAIN=""
```

Queste informazioni vengono inserite nel file /etc/default/lxc-net.

Dopo la creazione di file con le configurazioni necessarie, sarà possibile avviare il servizio con i comandi elencati di seguito.

```
systemctl enable lxc-net
systemctl start lxc-net
```

Dove il primo comando abilita lxc-net e dopo che questo è stato abilitato, con il comando successivo sarà possibile avviare l'esecuzione di lxc-net. Tale servizio verrà quindi eseguito all'avvio creando un bridge di rete per i nostri container. Questo aggiungerà inoltre, anche delle regole firewall attraverso l'utilizzo delle chiamate iptables.

4.6.3 Bridge Networking in LXD

LXD utilizza le funzionalità di networking di LXC. Per impostazione di default, esso connette i vari container in esecuzione sulla macchina host al dispositivo di rete lxcbr0. Qualora si avesse la necessità di utilizzare un'interfaccia differente da lxcbr0, bisognerà modificare l'impostazione di default utilizzando lo strumento della riga di comando lxc che ho descritto di seguito.

```
$ lxc profile edit default
```

In seguito un editor aprirà il file di configurazione il quale conterrà di default il seguente contenuto:

```
name: default
config: {}
devices:
  eth0:
    name: eth0
    nictype: bridged
    parent: lxcbr0
    type: nic
```

Settando il parametro parent, sarà possibile impostare qualunque bridge LXD che si desidera collegare ai container come impostazione di default. Di seguito descriverò un esempio, utilizzando un package di LXD che fornisce esempi di configurazione di rete, che sono presenti in /usr/share/lxd.

```
$ ln -s /usr/share/lxd/dnsmasq-lxd.conf /etc/dnsmasq-lxd.conf
$ ln -s /usr/share/lxd/systemd/system/dnsmasq@lxd.service
/etc/systemd/system/dnsmasq@lxd.service
$ ln -s /usr/share/lxd/netctl/lxd /etc/netctl/lxd
$ ln -s /usr/share/lxd/dbus-1/system.d/dnsmasq-lxd.conf
/etc/dbus-1/system.d/dnsmasq-lxd.conf
```

Nel passo successivo modificherò il profilo di default e sostituento lxcbr0 in parent, con lxd.

```
$ lxc profile edit default
```

A questo punto abiliterò e avvierò dnsmasq@lxd.service è netctl@lxd.service

Dalla versione 2.3 invece, LXD ha bloccato la sua dipendenza da LXC per il bridge networking, e si è spostato sull'utilizzarne uno proprio (lxdbr0). lxdbr0 di default non ha configurata alcuna sottorete ipv4/ipv6 e solo il traffico HTTP viene instradato tramite proxy attraverso la rete. Questo implica che non è possibile utilizzare SSH tra i container se si ha la configurazione lxdbr0 di default. Questo cambiamento è stato attuato al fine di evitare di acquisire sottoreti per gli utenti, perchè questo potrebbe causare rotture dei container, consentendo inolotre agli utenti di scegliere le proprie sottoreti.

4.6.4 Bridge Networking in Rkt

Rkt gestisce il networking grazie al Container Network Interface (CNI), il quale è uno standard proposto per la configurazione delle interfacce di rete per i container Linux. Nel contesto di CNI, il container si riferisce specificamente a un namespace di rete Linux. La combinazione di Rkt con CNI, fornisce la possibiltà di applicare diversi tipi di configurazione di rete dedicata ai pod, la quale è l'unità di esecuzione in rkt che può contenere al suo interno una o più applicazioni in esecuzione, ciascuna all'interno di uno specifico container. In rkt le reti possono essere configurate utilizzando il flag –net una o più volte all'atto dell'invocazione dell'esecuzione di rkt. L'opzione –net è impostata su una rete denominata che dovrebbe essere caricata da rkt quando il pod viene eseguito. Ripetendo l'opzione invece, è possibile creare un elenco di reti. Di segutio descriverò alcune delle modalità di rete previste e che sono settabili attraverso l'utilizzo di appositi nomi di rete dedicati a ciascuna modalità e forniti all' opzione –net. Sono dunque presenti tre nomi di rete che identificano le varie modalità. Il terzo nome identificherà la configurazione di default e può esser definito con -net o -net=default.

Nessuna connettività di rete: -net = none

Se si desidera isolare completamente un pod dalla rete, è possibile farlo utilizzando –net = none. Questa è una pratica di sicurezza utilizzata per limitare i vettori mediante i quali è possibile accedere a un pod quando un'applicazione non dispone di determinati requisiti di rete. In questo caso, il pod finirà con solo l'interfaccia di rete loopback.

```
$ sudo rkt run --interactive --net=none busybox-1.23.2-linux-amd64.aci
(...)
/ # ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
       link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
       inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
       inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
/ # ip route
```

```
/ # ping localhost
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.022 ms
```

In questo caso si ha solo l'interfaccia con l'indirizzo locale. Inoltre la risoluzione di localhost è abilitata in rkt per impostazione default, e verrà generato /etc/host minimo all'interno del pod, nel caso in cui non venga fornito dall'immagine.

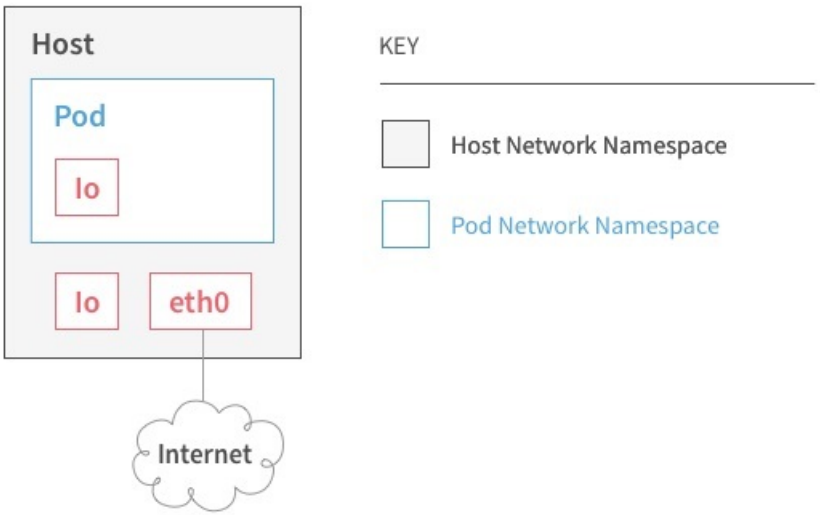

Figura 4.6. Exposed bridged network.

Rete host con nessun isolamento: -net=host

In questo caso rkt non richiede di isolare lo stack di rete del pod. L'accesso completo alla rete host può essere facilmente concesso a un pod selezionando l'opzione –net = host sulla riga di comando rkt. Questo però fa si che tutti i processi del pod condividano lo spazio dei nomi di rete dell'host. Di seguito confronterò gli spazi dei nomi di rete di un processo host e di un processo pod. E' possibile inoltre, controllare lo spazio dei nomi di processo effettuando un collegamento nei procfs.

```
$ readlink /proc/self/ns/net
net:[4026531969]
$ sudo rkt run --interactive --net=host busybox-
1.23.2-linux-amd64.aci --exec /bin/busybox
-- readlink /proc/self/ns/net

net:[4026531969]
```

Come mostrato di cui sopra, il pod non ha lasciato lo spazio dei nomi di rete dell'host, quindi potrà utilizzare la connettività di rete dell'host e potrà anche ascoltare su tutte le interfacce e le porte dell'host.

Default networking: -net=default

La rete di default invece, andrà ad utilizzare il plugin PTP CNI e per IPAM (IP Address Management) la rete utilizzerà il plugin host-local con un intervallo di indirizzi privato. Due fattori importanti per la connettività verso altre reti sono le impostazioni dei percorsi IPMasq e host-local di PTP. La mascheratura IP viene abilitata allo scopo di poter fornire la traduzione dell' indirizzo di rete che, insieme al percorso predefinito ("0.0.0.0/0" per IPv4), consente ai pod di

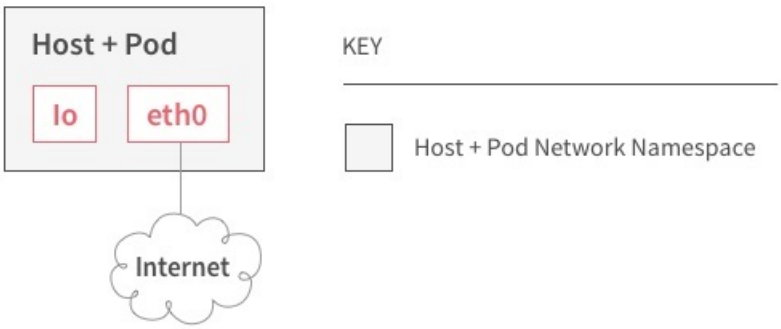

Figura 4.7. Exposed bridged network.

raggiungere qualsiasi altra rete con cui l'host è in grado di connettersi. Quindi, lasciando i pod nello spazio dei nomi di rete dell'host si avrebbero delle implicazioni riguardanti l'isolamento e la sicurezza.

La rete di default risulta inoltre il modo più rapido per consentire a un pod di connettersi a Internet. Di seguito eseguirò un'ispezione dall'interno del pod utilizzando la rete di default.

```
$ sudo rkt run --dns 8.8.8.8 --interactive
--debug busybox-1.23.2-linux-amd64.aci
(...)
2015/12/14 11:34:24 Loading network default with type ptp
(...)
/ # ip -4 address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
       inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>
 mtu 1500 qdisc noqueue
       inet 172.16.28.2/24 scope global eth0
       valid_lft forever preferred_lft forever
/ # ip -4 route
default via 172.16.28.1 dev eth0
172.16.28.0/24 via 172.16.28.1 dev eth0 src 172.16.28.2
172.16.28.1 dev eth0 src 172.16.28.2
```

Ora mi focalizzerò sulla descrizione delle implicazioni anticipate in precedenza.

Restricted Network Administration Capabilities L'utilizzo di –net = host con rkt non significa automaticamente che il pod può riconfigurare e modificare l'intero stack di rete. Infatti se ad esempio si cercasse di disattivare l'interfaccia di loopback, tale tentativo non andrà a buon fine, restituendo il seguente risultato.

```
/ # ip link set down dev lo
ip: SIOCSIFFLAGS: Operation not permitted
```

Nonostante sia UID0 e abbia eredito lo spazio dei nomi di rete dell'host, il pod non è in grado di modificare la configurazione di rete. Questo è dovuto al fatto che rkt limiti la capability CAP_NET_ADMIN.

Network namespace resources exposed Le applicazioni in esecuzione in un pod che condividono lo spazio dei nomi host sono in grado di accedere a tutto ciò che è associato alle interfacce di rete dell'host: indirizzi IP, percorsi, regole firewall e socket, incluse le socket UNIX astratte. A seconda della configurazione dell'host, le socket UNIX astratte, utilizzate da applicazioni come X11 e D-Bus, potrebbero esporre endpoint critici alle applicazioni del pod. Questo rischio può essere evitato configurando uno spazio dei nomi separato per i pod.

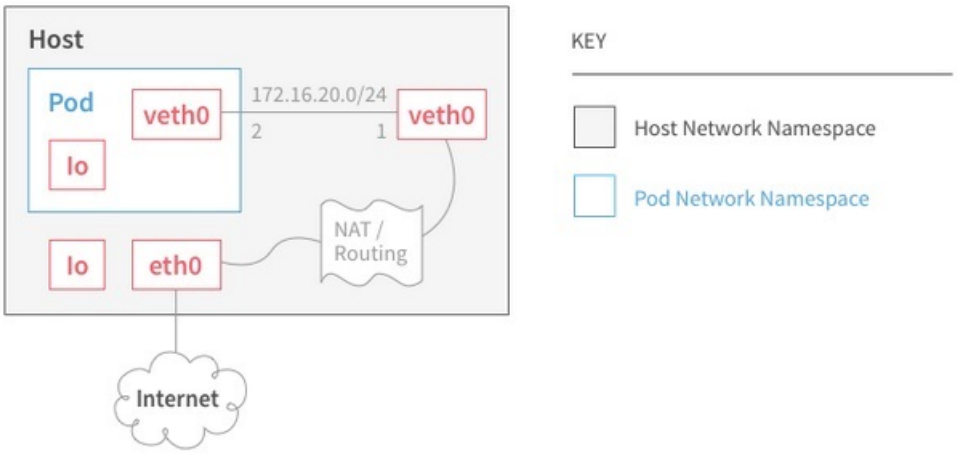

Figura 4.8. Exposed bridged network.

Nelle sezioni seguenti mostrerò come rkt utilizza CNI per impostazione di default e come è possibile personalizzarlo.

CNI - Plugin e configurazione In rkt, i binari del plugin CNI a monte sono incorporati e forniti da coreos, che è anche la scelta di default situata a monte. I plugin utilizzati per impostazione predefinita in rkt sono ptp con host-local e possono essere trovati nell'archivio rkt come file binari semplici:

```
$ sudo find /var/lib/rkt/cas/tree -regex ".*/\(host-local\|ptp\)"
 -exec dirname {} \; | uniq
/var/lib/rkt/cas/tree/deps-sha512-b0d71ba92df0764c60df72b43bc1
5b3da391733e55f918de17105ac6da4277eb
/rootfs/usr/lib64/rkt/plugins/net
```

Nonostante il lungo nome della directory prelevato dall'UUID del pod, è possibile vedere che i rootfs estratti sono comunque una gerarchia di file standard. Entrambi i plugin CNI sono nella directory net/, insieme agli altri plugin di rkt:

```
$ sudo ls -1 /var/lib/rkt/cas/tree/
deps-sha512-b0d71ba92df0764c60df72b43bc15b3da3917
33e55f918de17105ac6da4277eb/rootfs/usr/lib64/rkt/plugins/net
bridge
dhcp
flannel
host-local
ipvlan
macvlan
ptp
```

Questi binari vengono eseguiti da rkt dopo aver caricato i file di configurazione di rete, determinando quali eseguire e quali parametri vanno passati.

Rkt inoltre, può essere istruito allo scopo di poter impostare delle regole firewall per inoltrare i pacchetti su porte specifiche dell'host ai nodi designati. Le regole corrispondono alle porte specificate sugli indirizzi dell'host e sui pacchetti ricevuti all'indirizzo del veth del pod. Quindi se il file manifest elenca tali porte, con l'opzione -port utilizzata durante l'invocazione di rkt, sarà possibile configurare queste regole. Nell'esempio seguente utilizzerò un'immagine busybox contenente una porta nel manifest.

```
"ports": [
       {
```

```
                "name": "nc",
                "protocol": "tcp",
                "port": 1024,
                "count": 1,
                "socketActivated": false
        }
]
```

Nel comando successivo manderò in esecuzione il pod, reindirizzandolo alla porta TCP 1024 (denominata nc nel manifest), passando da tutti gli indirizzi host all'indirizzo veth del pod. Il pod si fermerà all'uscita di netcat, ovvero dopo che la connessione remota verrà chiusa. Per tale dimostrazione si andrà quindi ad utilizzare due terminali.

Nel primo terminale eseguirò i seguenti comandi, ricevendo come risultato quando descritto di seguito:

```
$ sudo rkt run --dns 8.8.8.8 --interactive --debug
 --port nc:1024 busybox-1.23.2-pfwd-linux-amd64.aci --exec
/bin/busybox -- nc -p 1024 -l
(..)
Starting Application=busybox Image=busybox...
[ OK ] Reached target rkt apps target.
```

Nel secondo terminale invece ho eseguito:

```
$ echo the host says HI | nc 172.16.28.1 1024
```

Questo implicherà che il terminale rkt restituisca il seguente messaggio per poi in seguito terminare.

```
the host says HI
Sending SIGTERM to remaining processes...
(..)
```

Mentre uno qualsiasi degli indirizzi dell'host potrebbe essere stato utilizzato per raggiungere il pod, questo esempio utilizza l'indirizzo IP dell'host sulla rete predefinita, che verrà associato anche alle regole firewall. Questo indirizzo è anche l'inizio della gamma di sottoreti 172.16.28.0/24 ed è quindi 172.16.28.1.

Al contrario della rete di default, la default restricted network non fornisce una connessione con il mondo esterno, rendendo le differenze tra le due configurazioni immediatamente evidenti:

```
--- ./stage1/net/conf/99-default.conf
2015-11-25 09:47:23.255816427 +0100
+++ ./stage1/net/conf/99-default-restricted.conf
 2015-11-25 09:47:23.255816427 +0100
@@ -1,12 +1,9 @@
 {
- "name": "default",
+ "name": "default-restricted",
        "type": "ptp",
- "ipMasq": true,
+ "ipMasq": false,
        "ipam": {
                "type": "host-local",
-  "subnet": "172.16.28.0/24",
-  "routes": [
-   { "dst": "0.0.0.0/0" }
-  ]
+  "subnet": "172.16.28.0/24"
        }
 }
```

Non avendo quindi nessuna maschera IP e nessun percorso predefinito, il pod non sarà in grado di comunicare oltre alla propria sottorete, che include solo l'estremità veth dell'host e gli altri pod nella rete di default. Inoltre, la default restricted network, verrà caricata automaticamente passando un nome di rete reale a –net; ovvero qualsiasi nome diverso da none, host e default.

Capitolo 5

Proposta di una nuova soluzione di sicurezza

In questo capitolo ci si focalizzerà sul presentare una nuova soluzione di sicurezza per i container Linux, incentrata sull'automatizzazione di policy SELinux e prendendo come esempio implementativo un ambiente basato su container Docker ed in esecuzione su una piattaforma avente CentOS 7 come distribuzione Linux e SELinux abilitato di default. Le versioni degli strumenti utilizzati sono indicate nel manuale utente 5.2

Poichè l'idea si basa sull'automatizzazione della sicurezza per un ambiente di container, anche l'esecuzione di questi ultimi andrà automatizzata, andando inoltre a recuperare le immagini necessarie all'implementazione di tale ambiente, da dei repository sicuri che nel caso dei container Docker sarà Docker Hub che si andrà ad appoggiare al DTR o *Docker Trusted Registry*, allo scopo di garantire l'integrità di ciascuna immagine. Questo è possibile solo se l'avvio di un container viene fatto tramite comando *docker run* o *docker pull* seguito dal primo comando, se si vuole prima scaricare l'immagine e poi la si vuole eseguire all'interno di un container. Questo passaggio come anticipato serve per garantire l'integrità dell'immagine stessa, infatti in caso contrario, un'immagine alterata da qualcuno che non è il proprietario della stessa, non verrà scaricata sul proprio sistema host. Un altro meccanismo che garantisce l'integrità di un'immagine quando questa andrà in esecuzione è il *docker-compose*, il quale permette inoltre di automatizzare l'esecuzione di più container i quali saranno in grado di comunicare tra di loro a meno di specifici vincoli. Questo tipo di automatizzazione viene eseguita tramite la lettura di un file .yaml il quale conterrà tutte le specifiche di avvio per ciascun container, come: l'immagine che dovrà essere eseguita, il volume ad essa dedicato, le risorse hardware quali cpu, memoria, memoria di swapping assegnabili a ciascun container e altri possibili attributi elencati in modo più esaustivo nella documentazione ufficiale di Docker.

5.0.1 Docker-compose e Docker run a confronto

Gli strumenti come Docker-compose e docker run vengono utilizzati da Docker allo scopo di mandare in esecuzione delle applicazioni all'interno di specifici container Docker. Il Docker run command, permette di eseguire l'immagine di una specifica applicazione la quale viene scaricata sulla macchina host attraverso dei repository sicuri che come anticipato nella parte introduttiva di questo capitolo, forniscono un certo meccanismo di integrità dell'immagine che si andrà ad utilizzare. A tale comando possono essere assegnati ulteriori campi che definiscono le caratteristiche e i vincoli che il container dovrà avere per poter ospitare una specifica immagine, come ad esempio gli attributi –memory, –memory-swap e –cpus al fine di limitare le risorse assegnabili al container, oltre ad attributi come –security-opt che permette di di gestire opzioni di sicurezza come ad esempio abilitare/disabilitare le funzionalità di Seccomp ed anche l'assegnazione di etichette di contesto SELinux, oltre agli attributi standard utilizzati per la definizione del nome del container, l'assegnazione di una specifica rete e l'attributo per definizione di una specifica porta.

Per una lista più dettagliata delle funzionalità e del comando docker run ho fatto riferimento alla documentazione ufficiale Docker [57]. Di seguito, un esempio di utilizzo del comando Docker run.

```
docker run -d --network network_name --security-opt
    label=type:svirt_lxc_net_t --security-opt label=level:s0:c0.c122 -p
    port_val1 --name container_name image_name
```

Il comando docker run permette quindi un'automatizzazione a livello di specifiche di configurazione per singolo container, ma non fornisce un meccanismo utile ad implementare in modo automatico un ambiente di container. Docekr-compose a differenza del precedente, permette la definizione delle specifiche di configurazione per-container e inoltre, automatizza anche il meccanismo di composizione di un ambiente di container. Questo tipo di procedura è attuato attraverso l'editing di un file .yaml il quale contiente tutte le specifiche necessarie all'avvio di un ambiente di container Docker. Attraverso l'utilizzo del comando docker-compose, verrà letto questo file di configurazione e verranno scaricate e mandate in esecuzione su differenti container, tutte le immagini necessarie alla messa in opera di uno specifico ambiente, andando a settare inoltre, se precedentemente definito all'interno del file di configurazione .yaml la rete tra i container, gestendo inoltre la comunicazione tra di essi. Ovvero chi è abilitato a comunicare con chi e a quali altri gruppi di container è negata la comunicazione diretta tra loro.

Di seguito, un esempio di utilizzo del comando Docker run.

```
sudo docker build -t direcory_name .
docker-compose up
```

Poichè l'utilizzo di tale comando vada a buon fine è necessario che venga eseguiuto da terminale all'interno della directory dove è presente il file di configurazione, che nel caso indicato di cui sopra prende il nome di direcotry_name [58]. La presenza di un ulteriore file che prende il nome di Dockerfile e il quale viene definito senza alcuna estensione, permette di inserire particolari specifiche che garantiscono la riproducibilità di uno specifico ambiente in qualunque altro sistema[59].

La descrizione di questi due metodi di avvio di container Docker è stata utile allo scopo di poter comprendere con maggior chiarezza l'approccio utilizzato per lo sviluppo di uno script che permette l'automatizzazione della sicurazza tra i container Docker.

Infatti, prendendo in considerazione l'automatizzazione di un ambiente di container tramite l'utilizzo del comando docker-compose e del *docker-compose* file, risulta possibile assegnare dei contesti SELinux per ciascun container. Questo è utile se l'ambiente che si andrà ad eseguire, sarà costituito da container i quali eseguiranno dei processi che dovranno andare a modificare o leggere degli specifici file all'interno di specifiche directory. Quindi si potrebbe volere che uno dei processi sia abilitato a leggere e scrivere i file di una specifica sotto cartella e che l'altro processo invece, possa leggere e scrivere i file di un'altra sotto cartella appartenente alla stessa directory della precedente, ma che non possa essere fatto il contrario. Una soluzione è quella di andare ad impostare il quarto campo del contesto SELinux assegnando livelli differenti a ciascun container e assegnando lo stesso livello alla stessa directory che per il quale si vuole che il container abbia accesso. Un altro caso si ha quando ad esempio si vuole evitare che un container sotto attacco non acceda alle directory del sistema host. Infatti anche se il processo venisse eseguito all'interno di un container, l'isolamento con la macchina host non è completamente garantito. Quindi l'applicazione di etichette di contesto differenti da quelle dei container risulta una soluzione di sicurezza aggiuntiva. Prendendo in considerazione Docker, questo avvia direttamente i container assegnando un SELinux type uguale per tutti, ma differente dal SELinux type assegnato agli altri oggetti in esecuzione sulla macchina host, in modo da garantire e fornire un ulteriore livello di isolamento tra container e sistema sottostante 5.1.

Se invece i container che andranno in esecuzione dovranno comunicare all'interno di una rete Docker a loro dedicata, questo approccio non è più utile. Si deve pensare quindi a come isolare specifici container tra loro e permettere la comunicazione con altri, avvalendosi dello strumento di sicurezza SELinux. Un approccio iniziale poteva essere quello di andare a gestire la rete Docker dedicata a tale ambiente di container multipli, per poi in seguito etichettare porte, indirizzi ip

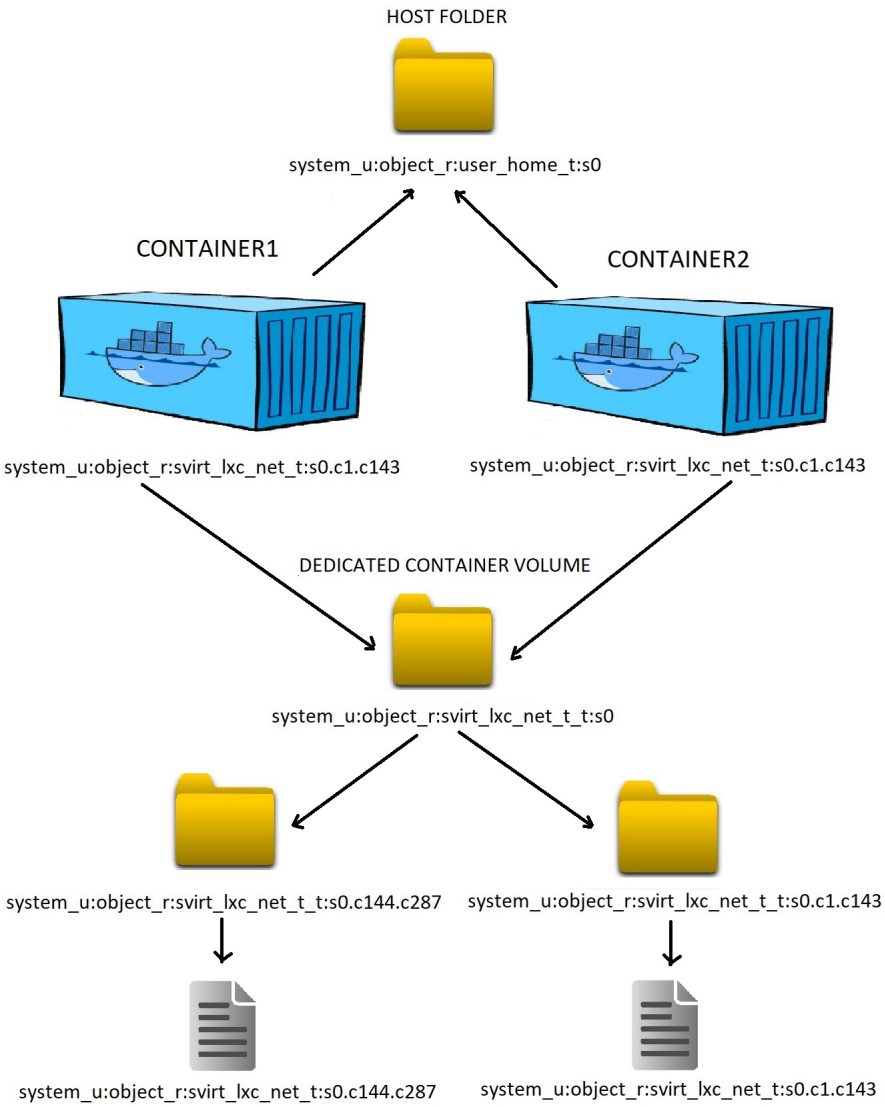

Figura 5.1. SELinux prevent access to file and directory.

e interfacce di rete. Questo non ha portato a dei buoni risultati, benchè sia possibile assegnare etichette alle reti del sistema host, non lo è stato per le reti generate da Docker.

Un secondo approccio, il quale ha portato a differenti risultati è stato quello di andare a gestire le iptables del sistema host, ovvero assegnare delle regole *firewall* che permettano di gestire l'inoltro dei pacchetti da una sorgente e dirette verso una specifica destinazione, attraverso la gestione degli indirizzi IP ad essi associati 3.1.9. A questo punto si può pensare ad un ambiente implementativo costituito da quattro container e di cui ognuno avvierà una delle immagini indicate di seguito.

```
Image1:haproxy load-balancer
Image2:python application service 1
Image3:python application service 2
Image4:database mysql
```

I quattro container apparterranno ad una rete Docker dedicata e la comunicazione tra di essi dovrà essere instaurata in modo tale che il primo container il quale manderà in esecuzione *Haproxy* e che dovrà funzionare come *Reverse-Proxy* e *Load-Balancer*, potrà comunicare con i due *Application-Services*, andando a bilanciarne il traffico, ma non potrà comunicare direttamente, con il container

che manderà in esecuzione il database mysql. La comunicazione con il database, dovrà essere concessa solo ai due *Application-Services* e questa comunicazione dovrà essere bidirezionale. Il container che eseguirà *Mysql*, non dovrà essere in grado di comunicare a sua volta con *l'Haproxy* container. 5.3 Questo tipo di configurazione può esser definito come detto in precedenza, mediante l'applicazione di regole di inoltro dei pacchetti, all'interno delle *iptables*.

All'atto della definizione di tali regole, verrà utilizzato l'attributo *SECMARK*, anzichè gli attributi standard *ACCEPT* e *DROP*, per la gestione dell'inoltro dei pacchetti, allo scopo di poter gestire tali regole, mediante policy SELinux.

Nell'esempio implementativo che è stato appena descritto, risulta necessario bloccare il traffico tra *L'Haproxy* container ed il container che eseguirà il database *Mysql*, andando quindi a vietare l'inoltro dei pacchetti in modo bidirezionale. Di seguito sono indicati i comandi che andranno ad imporre tali regole all'interno delle *iptables*.

```
sudo iptables -A FORWARD -t mangle -s 172.18.0.5 -d 172.18.0.2 -j SECMARK
    --selctx system_u:object_r:haproxy_to_db_packet_t:s0

sudo iptables -A FORWARD -t mangle -s 172.18.0.2 -d 172.18.0.5 -j SECMARK
    --selctx system_u:object_r:db_to_haproxy_packet_t:s0
```

Le etichette *haproxy_to_db_packet_t* e *db_to_haproxy_packet_t* vengono definite a priori mediante policy SELinux come quella descritta di seguito.

```
policy_module(secmark, 1.0)

gen_require('attribute svirt_lxc_net_t;')

#Type Packet and Rules Definitions
type haproxy_to_db_packet_t;
dontaudit svirt_lxc_net_t haproxy_to_db_packet_t:packet { send };

type db_to_haproxy_packet_t;
dontaudit svirt_lxc_net_t db_to_haproxy_packet_t:packet { send };
```

Tale policy definisce che a tutti i pacchetti che sono stati etichettati come *haproxy_to_db_packet_t* e che sono stati generati da processi il cui SELinux type è *svirt_lxc_net_t*, deve essere negato l'inoltro. Lo stesso vale anche per i pacchetti in uscita dal *Mysql* container e diretti verso *l'Haproxy* container, di cui tale vincolo è applicato dalla seconda regola definita all'interno della Policy SELinux.

Questo meccanismo garantisce dunque la corretta assegnazione dei vincoli precedentemente imposti. A questo punto si può passare all'automatizzazione di tale procedura, in modo che questo possa essere applicato a qualunque tipo di ambiente di container Docker.

E' stato quindi sviluppato uno script in linguaggio Python, il quale andrà a leggere un qualunque file *docker-compose.yml* contenente le specifiche per l'implementazione di uno specifico ambiente di container.

Lo script da me sviluppato dopo la fase di lettura del file *.yaml*, andrà inizialmente a creare una rete Docker dedicata ai contanier, per poi in seguito eseguire dei comandi di *Docker run* per ciascuna immagine indicate nel file precedentemente menzionato. Il comando di *Docker run* andrà quindi ad eseguire un'immagine all'interno di un container definito da un nome univoco, tale container sarà assegnato alla rete precedentemente definita e sarà etichettato mediante uno specifico contesto SELinux. Inoltre a ciascun container verrà assegnato un volume specifico e se presenti nelle specifiche del file *yaml*, verranno applicate delle specifiche limitazione delle risorse hardware ad esso assegnate, come la limitazione di risorse *CPU*, *memoria Ram* e *memoria di swapping*.

Come illustrato in figura 5.2 non è stato utilizzato il comando di docker-compose per la generazione dell'ambiente di container, attraverso la lettura di un file .yaml, poichè tale comando

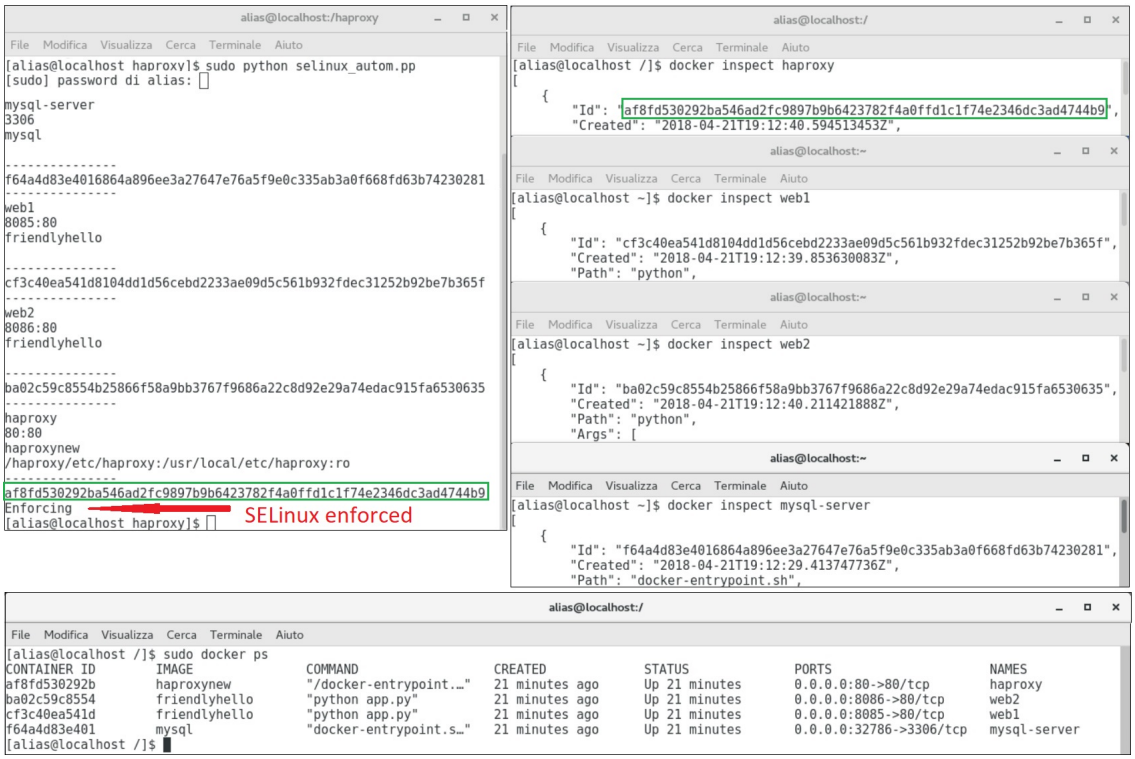

Figura 5.2. Esecuzione algoritmo di automazione della sicurezza nei container Linux.

non permette la generazione automatica di policy SELinux e di regole firewall da assegnare all'interno delle iptables allo scopo di gestire le comunicazioni tra i container e la macchina host. Quindi per una completa automatizzazione, si è scelto di utilizzare il comando di docker run il quale attraverso lo script da me sviluppato, verrà popolato ad-hoc dagli attributi necessari alla generazione di uno specifico container, i quali sono definiti all'interno del file .yaml. Il meccanismo implementativo, consiste essenzialmente nella dichiarazione di parte degli attributi che verranno generalmente utilizzati da tale comando e nella definizione di flag associati a ciascuno di essi. Se all'atto della generazione del comando docker run saranno necessari gli attributi come –network e ad esempio –security-opt, i flag di questi saranno attivati, e quindi solo questi due attributi saranno utilizzati per generare il comando di docker run per uno specifico container. Un esempio di tale meccanismo di automazione è presentato di seguito.

```
subprocess.call(['sudo','docker','run','-d','--network',
network_name,'--security-opt',
'label=type:svirt_lxc_net_t','--security-
opt','label=level:s0:c'+str(c1)+'.c'+str(c2),'-p',port_val1,'-
v',volumes_name,'--name',container_name,image_name])
```

Gli attributi che invece non sono stati menzionati, come ad esempio -p e -v non verranno inseriti all'atto della generazione del comando. Il comando subprocess.call(...) viene utilizzato in python allo scopo di poter automatizzare le chiamate effettuate da riga di comando. E' necessario per il corretto funzionamento, che sia presente il seguente import all'interno del codice.

```
import subprocess
```

Al passo successivo, verranno generate delle policy SELinux che definiranno le etichette per i pacchetti, le quali andranno assegnate mediante regole firewall imposte all'interno delle iptables. I container che comunicano tra di loro sono quelli che presentano l'attributo *Dependency* presente all'interno del file *yml* e seguito dal nome del container. Il resto delle comunicazione tra container dovrà essere bloccato. Quindi sulla base di questi vincoli, andranno definite le policy SELinux e le regole firewall all'interno delle iptables.

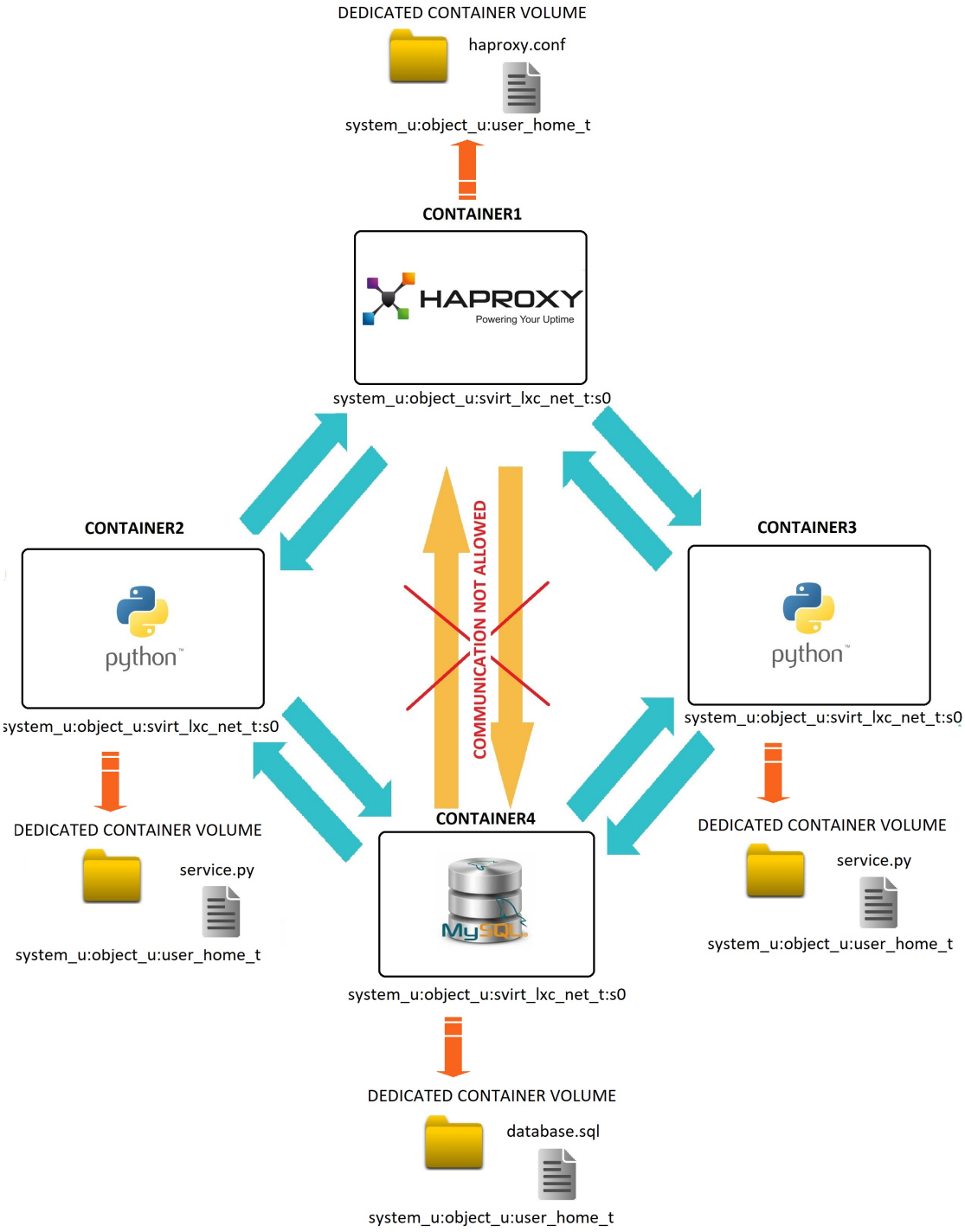

Figura 5.3. Architettura costituita da un ambiente di container Docker per implementazione di policy SELinux.

Uno schema a blocchi illustrato nella figura 5.4 indica i passi svolti dall'algoritmo da me sviluppato allo scopo di automatizzare la sicurezza in ambito container.

La figura 5.5 illustra i risultati del corretto funzionamento dell'esempio di ambiente di container multipli da me implementato, mostrando inoltre il corretto funzionamento dell' haproxy il quale esegue *load-balancing* per il bilanciamento del traffico attraverso l'utilizzo dell'algoritmo *Round-Robin*.

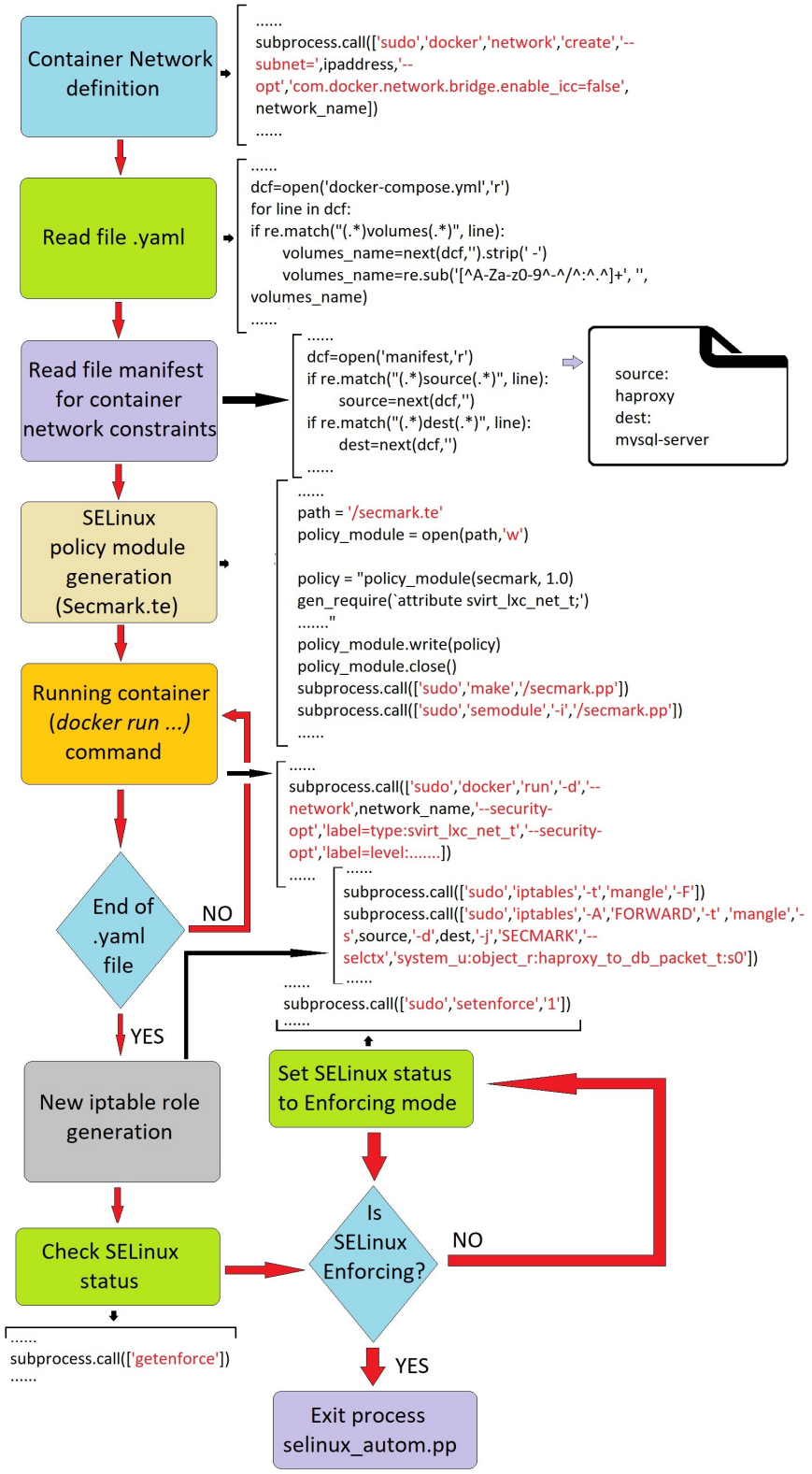

Figura 5.4. Schema a blocchi algoritmo di automazione.

Prima di passare alla descrizione dei risultati ottenuti effettuerò una breve introduzione di un'altra soluzione di sicurezza sempre basata su policy SELinux che però è applicabile anche ad ambienti costituiti da macchine differenti che comunicano tra loro attraverso la rete, attraverso lo

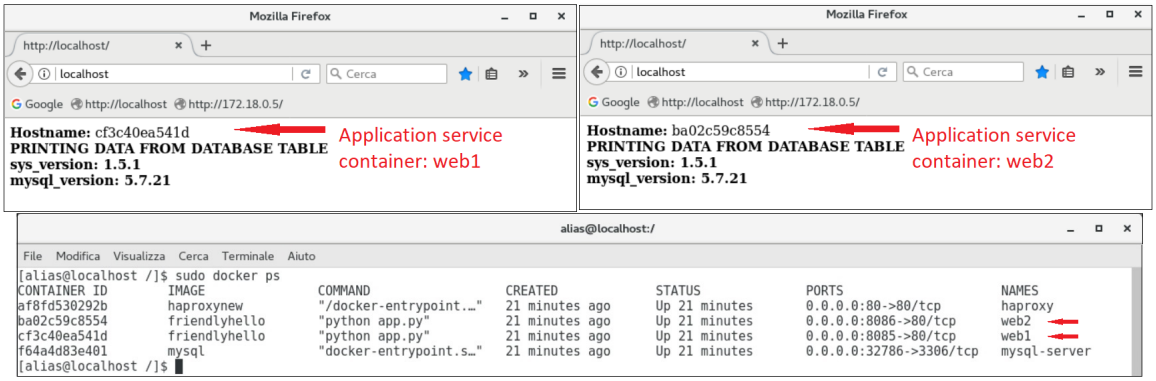

Figura 5.5. Test ambiente di container

strumento di sicurezza Labeled-IPsec 3.1.10. Un esempio di possibile implementazione potrebbe essere quello di utilizzare un browser interno ed uno esterno sul sistemi aziendali dei dipendenti. Il browser configurato come *interno*, verrà eseguito all'interno di un dominio a cui è consentito accedere ai server di applicazioni web aziendali interne che contengono informazioni riservate sui vari clienti, mentre il browser *esterno* potrà accedere in remoto e quindi ad internet. Questo riduce il rischio che i contenuti internet illegali possano andare a compromettere i dati interni alla macchina con conseguente compromissione virale delle altre macchine collegate alla rete aziendale. Questo tipo di separazione, risulterebbe invece molto più difficile se non impossibile, senza i controlli avanzati di rete di SELinux. Nella figura 5.6 viene illustrato un esempio implementativo di tale soluzione.

5.1 Manuale programmatore

Lo script sviluppato con lo scopo di automatizzare l'esecuzione di un ambiente basato su container multipli insieme all'automatizzazione di regole firewall e generazione automatica di policy SELinux, verrà descritto in modo specifico con lo scopo di comprenderne meglio l'architettura 5.7 ed il suo funzionamento.

Tale script sviluppato in linguaggio Python è presente nel disco che è stato consegnato alla commissione di laurea con il nome di:

```
yaml_reader.py
```

Di seguito è stata analizzato ogni blocco di codice contenuto all'interno di tale file.

Il primo blocco di codice indicato di seguito è stato sviluppato con lo scopo di andare ad assegnare uno specifico range di indirizzi ip per ciascun ambiente di container. Il meccanismo consiste nell'andare a leggere un file identificato come /network.txt il cui contenuto sarà costituito da un numero che verrà utilizzato per generare un primo range di indirizzi ip da associare al primo ambiente di container. A seguito della lettura di tale numero, questo verrà sostituito in modo incrementale, in modo tale che nel momento in cui si andrà a richiamare lo script per generare un nuovo ambiente di container, gli possa essere assegnato un range di indirizzi ip differente dal precedente e così via ogni qual volta risulterà necessario utilizzare tale algoritmo per la generazione di ambienti di container.

Definizione del percorso del file per l'assegnazione dinamica degli indirizzi ip:

```
path = 'network.txt'
```

Apertura e lettura del file:

```
level_list = open(path,'r')
subnet=level_list.read()
level_list.close()
```

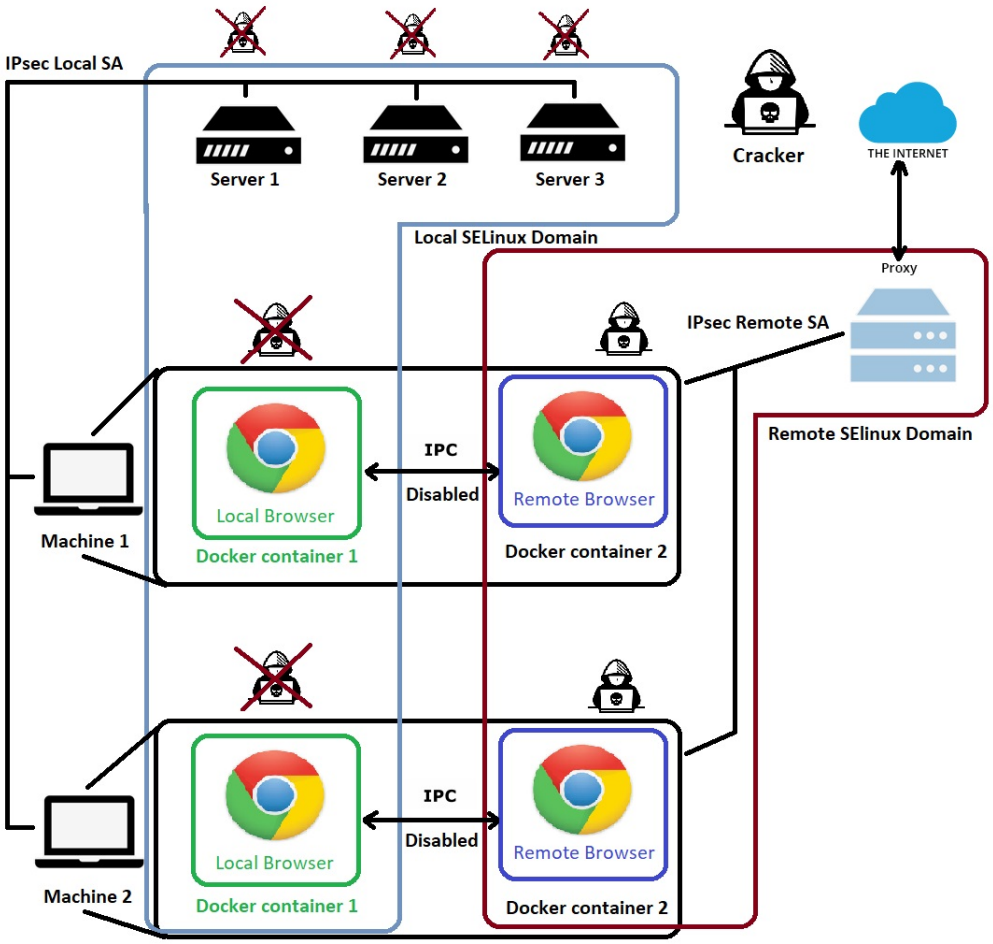

Figura 5.6. Labeled IPsec Environment. Da questa illustrazione si evince come un utente malevolo possa essere limitato attraverso Labeled IPsec al solo dominio remoto di SElinux, e quindi al proxy, e ai container dove vengono mandati in esecuzione i browser per accesso remoto. In aggiunta è stato anche disabilitato il meccanismo di Inter process Communication.

Definizione di una variabile il cui contenuto sarà costituito dal vecchio valore letto da file ed incrementato di una unità.

```
new_subnet=str(subnet+1)
```

Apertura del file in scrittura per l'inserimento del valore aggiornato il quale sarà utile per l'avvio di un nuovo ambiente di container.

```
level_list = open(path,'w')
level_list.write(str(new_subnet))
level_list.close()
```

Definizione nome di rete dinamico sulla base del valore di tipo intero letto dal file network.txt

```
network_name = 'environment_network_'+subnet
```

Il blocco di codice seguente permette di automatizzare i comandi docker i quali altrimenti andrebbero inseriti manualmente da terminale.

```
subprocess.call(['docker','network','create','--subnet='+str(subnet),
'--opt','com.docker.network.bridge.enable_icc=false',network_name])
```

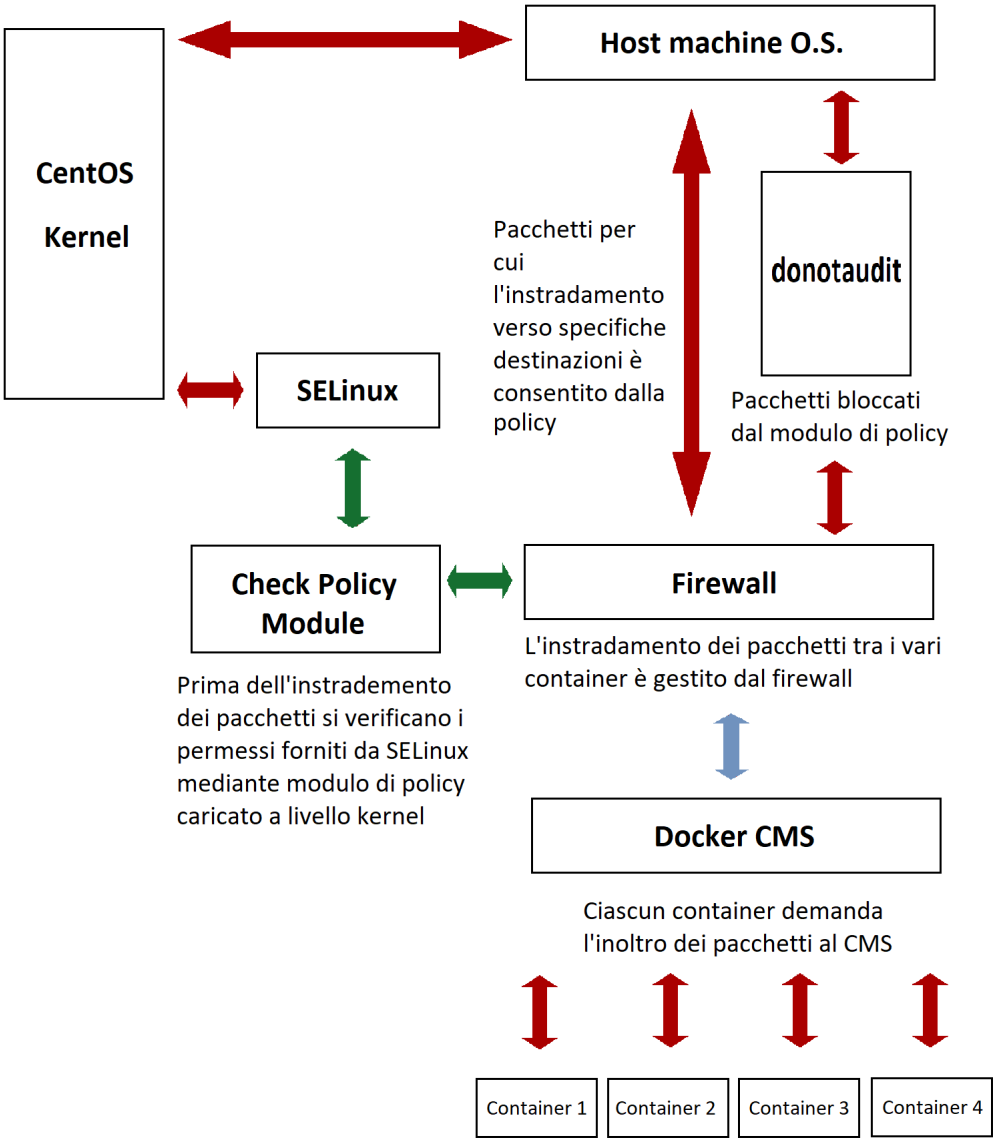

Figura 5.7. Architettura script automatizzazione container.

Il comando docker rappresentato di cui sopra, si occupa quindi di settare a *false* e quindi di disabilitare il meccanismo di *Inter-Container-Communication* (ICC) fornito da Docker. Questo passo è risultato utile in quanto l'abilitazione del meccanismo di ICC si va a sovrapporre alle regole *firewall* assegnate per ciascun container e definite all'interno delle *iptables* del sistema host, rendendo quindi inutile l'applicazione di tali regole. Per evitare le collisioni di indirizzi ip risulterà però necessario che l'esecuzione dei container venga effettuata solamente mediante lo script di automazione, in quanto l'utilizzo di tale script, come descritto in precedenza, andrà a mitigare tale problema. Nel blocco appena descritto sono stati però riscontrati dei problemi relativi al funzionamento di tale meccanismo e si è scelto quindi di adottare una soluzione statica basata sulla definizione manuale della rete da associare all'ambiente di container preso in considerazione come caso di studio. E' stato dunque inserito da riga di comando del terminale il comando seguente:

```
docker network create --subnet=172.18.0.0/16 --opt
    com.docker.network.bridge.enable_icc=false network_environment1
```

L'attributo *network_environment1* rappresenta il nome assegnato alla rete dedicata all'ambiente di container preso in considerazione come caso di studio.

Mediante la lettura e scrittura di un file definito ed inizializzato a priori con un intero di valore pari a 0, si andrà ad assegnare un livello differente di contesto SELinux a ciascun processo ed oggetto al quale tale processo avrà accesso. Il file in questione è identificato con il nome di /context_level_list.txt e risulta anch'esso presente nella directory principale del disco menzionato in precedenza. Definizione percorso del file:

```
path = 'context_level_list.txt'
```

Meccanismo di lettura del file:

```
level_list = open(path,'r')
level_from_file=level_list.read()
level_list.close()
```

Definizione variabile di tipo intero che acquisirà il numero letto dal file:

```
c1=int(level_from_file)
```

Definizione variabile aggiornata sulla base del numero precedentemente letto da file:

```
c2=int(c1)+143
```

Apertura del file in scrittura per la la sostituzione del nuovo valore con il vecchio:

```
level_list = open(path,'w')
```

Scrittura e chiusura del file:

```
level_list.write(str(c2+1))
level_list.close()
```

Il meccanismo di funzionamento del codice appena rappresentato e descritto, consiste essenzialmente nel generare in modo sequenziale un intervallo di numeri e salvarli all'interno di un comune file di testo chiamato /context_level_list.txt, in modo da garantire che SELinux associ un intervallo differente relativo al quarto campo *"level"* dell'etichetta di contesto per ciascun container. Questo è stato pensato con lo scopo di evitare che un processo in esecuzione all'interno di uno specifico container possa accedere a file o directory associate ad un altro container.

La parte di codice descritta di seguito invece automatizza l'esecuzione dei container mediante la chiamata del comando *docker-run* dopo la lettura del file /docker-compose.yml 5.12 contenente le specifiche di configurazione per ciascun container.

Apertura in sola lettura, del file /docker-compose.yml:

```
dcf=open('docker-compose.yml','r')
```

Definizione ed inizializzazione delle variabili che verranno utilizzate per l'interpretazione del file /docker-compose.yml:

```
image_name=''
container_name=''
port_val1=''
port_val2=''
port_val=''
environment = 'MYSQL_ROOT_PASSWORD=secret'
```

Lettura ed interpretazione del file:

```
def after(value, a):
    pos_a = value.rfind(a)
    if pos_a == -1: return ""
    adjusted_pos_a = pos_a + len(a)
    if adjusted_pos_a >= len(value): return ""
    return value[adjusted_pos_a:]
```

```
for line in dcf:
```

La parte seguente andrà a ricercare un eventuale match con la stringa *container_name* e se presente, si andrà a leggere il nome del container in questione per poi inserirlo all'interno della variabile *container_name*.

```
if re.match("(.*)container_name(.*)", line):
    container_name=line.split(':')[1].strip(' ')
    container_name =re.sub('[^A-Za-z0-9^-]+', '', container_name)
```

Al passo successivo si andrà a ricercare un eventuale match con la stringa *ports* e se presente, si andrà a leggere il nome del container in questione per poi inserirlo all'interno della variabile *port_val1* e *port_val2*. Un esempio di definizione di 2 porte può essere quello di voler mappare la porta 80 nel container alla porta 8080 nell'host. Questo meccanismo permette di non esporre le porte del container, direttamente all'esterno del sistema host e quindi in remoto.

```
if re.match("(.*)ports(.*)", line):
    port_val=next(dcf,'').strip(' -')
    port_val1=port_val.split(':')[0]
    port_val1=''.join(e for e in port_val1 if e.isalnum())
    port_val2=(after(port_val,":"))
    port_val2=''.join(e for e in port_val2 if e.isalnum())
```

La parte di codice rappresentata di seguito cercherà in un modo analogo a prima, una corrispondenza con la stringa *image*.

```
if re.match("(.*)image(.*)", line):
    image_name=str(line.split(':')[1].strip(' '))
    image_name=''.join(e for e in image_name if e.isalnum())
```

La riga di codice seguente è costituita da un *if* che controllerà se per uno specifico container sono state definite una o 2 porte, ovvero se tale container avrà come specifiche di configurazione, la necessità di mappare una specifica porta all'interno di un container con una porta sull'host.

```
if port_val1!=''and port_val2!='' and container_name!='' and image_name
    !='':
```

Mediante controllo di *if* ed *else* si andrà a verificare se il container che dovrà essere mandato in esecuzione, sia o meno un container di tipo *database*. Questo è utile, poichè il database si avvierà con un certo ritardo rispetto agli altri. Quindi i container i quali necessiteranno dell'interazione con il *database* dovranno a loro volta attende il completo avvio di quest'ultimo. Per tale motivo verrà applicato un ritardo dopo l'avvio del database, in modo da ritardare il ciclo utili all'avvio dei successivi container.

```
    if image_name=='mysql':
```

Parte di codice che automatizza il comando di *docker run* per l'avvio dei container:

```
      subprocess.call(['sudo','docker','run','-d',
      '--network',network_name,'--security-opt',
      'label=type:svirt_lxc_net_t','--security opt',
      'label=level:s0:c144.c287','-p',port_val1+':
      '+port_val2,'--name',container_name,'-e',environment,image_name])
      time.sleep(10)
    else:
```

Parte di codice che automatizza il comando di *docker run* per l'avvio dei container:

```
        subprocess.call(['sudo','docker','run','-d','--network',
        network_name,'--security-opt',
        'label=type:svirt_lxc_net_t','--security-opt',
        'label=level:s0:c'+str(c1)+'.c'+str(c2),'-p',port_val1+':
        '+port_val2,'--name',container_name,image_name])
```

Reinizializzazione delle variabili utilizzate:

```
container_name=''
port_val1=''
port_val2=''
image_name=''
```

Sezione *if else* dedicata all'esecuzione di container costituiti da una sola porta di rete come ad esempio il database mysql il quale non avrà necessità di avere una porta mappata con una dell'host, in quanto a tale database vi accedono solo i container appartenenti alla stessa sottorete.

```
if port_val1!='' and port_val2=='' and container_name!='' and image_name
    !='':
   if image_name=='mysql':
      subprocess.call(['sudo','docker','run','-d',
      '--network',network_name,'--security-opt',
      'label=type:svirt_lxc_net_t','--security-opt',
      'label=level:s0:c144.c287',
      '-p',port_val1,
      '--name',container_name,'-e',environment,image_name])
      time.sleep(10)
   else:
      subprocess.call(['sudo','docker','run','-d',
      '--network',network_name,'--security-opt',
      'label=type:svirt_lxc_net_t','--security-opt',
      'label=level:s0:c'+str(c1)+'.c'+str(c2),
      '-p',port_val1,'--name',container_name,image_name])
```

Reinizializzazione delle variabili utilizzate:

```
container_name=''
port_val1=''
port_val2=''
image_name=''
```

Il blocco di codice seguente definisce il percorso del file /restriction.txt il quale è sempre presente nella directory principale del disco in dotazione. Tale file verrà quindi letto ed interpretato dallo script con lo scopo di acquisire le specifiche di restrizione che andranno applicate all'ambiente di container.

```
path = 'restriction.txt'
```

L'inizializzazione della variabile indicata di seguito è stata utile a generare il campo *"type"* di un'etichetta di contesto SELinux, in modo dinamico, con lo scopo di poterlo applicare in modo univoco a ciascuna regola firewall. Tale variabile, sempre per le medesime ragioni, verrà applicata anche al nome di policy SELinux che verrà generato, ed il quale sarà associato a ciascuna regola firewall con lo scopo di evitare un conflitto tra i nomi delle policy.

```
environment_number=0
```

lettura del file :

```
with open(path, "r") as ins:
   for line in ins:
```

Mediante scomposizione di ciascuna riga del file in quattro colonne, verranno letti attributi come il nome ed il numero del container primario e secondario, utili alle definizione delle regole firewall che verranno generate in seguito.

```
cnt_name1=line.split(':')[0]
cnt_numb1=line.split(':')[1]
cnt_name2=line.split(':')[2]
cnt_numb2=line.split(':')[3]
```

Il file /restriction.txt dovrà trovarsi nella stessa directory in cui è presente il file di automatizzazione /yaml_reader.py. Tale file sarà inoltre costituito da una o più righe strutturate nel seguente modo:

```
nome_container1:numero_container1:nome_container2:numero_container2:
```

gli attributi numero_container 1 e 2 sono utili per l'assegnazione dinamica degli indirizzi ip. Definizione del modulo di policy associato all'ambiente di container sulla base delle restrizioni precedentemente acquisite:

```
subprocess.call(['sudo','make','secmark'+str(environment_number)+'.te'])
path = '/secmark'+str(environment_number)+'.te'
```

Apertura del file in scrittura:

```
policy_module = open(path,'w')
```

Di seguito è stata indicata la stringa utilizzata per rappresentare il contenuto del modulo di policy, la quale è costituita da comandi utili a portare la limitazione delle chiamate di sistema associate a ciascuna etichetta di contesto SELinux.

```
policy = "policy_module(secmark"+str(environment_number)+", 1.0)
    gen_require('attribute svirt_lxc_net_t;') type
    context_type"+str(environment_number)+"_t; dontaudit svirt_lxc_net_t
    context_type"+str(environment_number)+"_t:packet { recv send };"
```

Scrittura e chiusura del file di policy:

```
policy_module.write(policy)
policy_module.close()
```

Generazione delle componenti utili alla messa in esecuzione del modulo di policy:

```
subprocess.call(['sudo','make','secmark'+str(environment_number)+'.if'])
subprocess.call(['sudo','make','secmark'+str(environment_number)+'.fc'])
subprocess.call(['sudo','make','secmark'+str(environment_number)+'.pp'])
```

Avvio del modulo di policy:

```
subprocess.call(['sudo','semodule','-i','secmark'+str(environment_number)+'.pp'])
```

L'utilizzo di *sudo* è risultato necessario al fine di poter disporre dei permessi per la generazione di tale modulo di policy. In caso di avvio senza permessi di super user, verrà riscontrato l'errore indicato di seguito.

```
libsemanage.semanage_create_store: Could not read from module store, active
    modules subdirectory at /etc/selinux/targeted/active/modules. (Permission
    denied).
libsemanage.semanage_direct_connect: could not establish direct connection
    (Permission denied).
semodule: Could not connect to policy handler
```

In sintesi, il codice analizzato di cui sopra ha permesso dunque di creare delle *policy* SELinux mediante la lettura di un file contenente le restrizioni per l'ambiente di container che dovrà essere mandato in esecuzione. Le *policy* verranno quindi generate con lo scopo di gestire i permessi associati alle etichette di contesto definite per le regole *firewall*.

Di seguito è stato indicato il codice necessario alla generazione delle regole firewall:

```
subprocess.call(['sudo','iptables','-A','FORWARD','-t' 'mangle',
'-s 172.18.0.'+cnt_numb1,'-d 172.18.0.'+cnt_numb2,'-j' 'SECMARK',
'--selctx','system_u:object_r:context_type'
+str(environment_number)+'_t:s0'])
```

Per la regola firewall indicata di cui sopra, è stata associata una particolare etichetta di contesto SELinux. Tale etichetta è stata definita nel primo blocco di codice dello script. Inoltre è risultato necessario l'utilizzo di *sudo* in quanto servono permessi di amministratore del sistema per la gestione delle *iptables*. L'utilizzo del comando di iptables senza essere preceduto da *sudo* ha portato all'errore seguente:

```
iptables v1.4.21: can't initialize iptables table mangle':  Permission denied
    (you must be root)
```

La stringa di codice definita di seguito incrementa la variabile descritta in precedenza 5.1.

```
environment_number=environment_number+1
```

Prima del completamento della procedura di automatizzazione verrà richiamato il comando per settare SELinux in modalità di *enforcing* la macchina host, nel caso in cui questo non fosse stato abilitato in precedenza. Inoltre, per una maggiore chiarezza verrà stampato a video lo stato attuale di SELinux.

```
subprocess.call(['sudo','setenforce','1'])
subprocess.call(['getenforce'])
```

5.2 Manuale utente

In questa sezione verrà descritta la procedura utile a poter eseguire in modo corretto lo script identificato con il nome di /yaml_reader.py, il quale è stato sviluppato con lo scopo di automatizzare l'avvio di container multipli, oltre che ad automatizzare la generazione di regole firewall e policy SELinux per la gestione del traffico di rete tra i vari container. L'esecuzione di tale script è stata fatta utilizzando una macchina host con sistema operativo Linux CentOS 7. Sono stati inoltre necessari software come Docker per la generazione dei container, Python per poter eseguire lo script di automatizzazione e SELinux installato per la gestione delle policy di sicurezza. Come indicato nel manuale programmatore, il file /yaml_reader.py è presente nella directory principale del disco messo in allegato con la tesi. Di seguito sono indicate le versioni della componenti software utilizzate:

- Sistema operativo CentOS 7
- Versione kernel 3.10.0-693.2.2.eL7.x86_64
- Python versione v.3.6
- SELinux Kernel Policy version 28
- Docker Container Mangement System versione v.17.02
- Docker compose versione 1.18.0 build 8dd22a9

Di seguito sono stati citati i riferimenti alle guide per l'installazione del software necessario alla messa in opera dell'ambiente di sviluppo.

- Guida all'installazione di CentOS 7 [64]
- Comando per la verifica della versione del kernel del sistema operativo: *uname -r*
- Guida all'installazione delle componenti utili all'esecuzione di SELinux per CentOS [66]
- Guida all'installazione di Python versione 3 per CentOS 7 [67]
- Guida all'installazione di Docker per CentOS [65]
- Guida all'installazione di Docker Compose per la generazione di immagini create ad-hoc.

Una volta installato Docker, si proseguirà con l'installazione di Docker Compose. Prima di tutto andrà installato il repository EPEL eseguendo il comando seguente:

```
# yum install epel-release
```

Di seguito il comando per l'installazione di un'ulteriore componente:

```
# yum install -y python-pip
```

A questo punto risulterà possibile installare Docker compose eseguendo il seguente comando:

```
# pip install docker-compose
```

Risulterà necessario anche effettuare un upgrade di tutti i packages Python installati su CentOS 7:

```
# yum upgrade python*
```

Ora sarà possibile controllare la versione del Docker compose con il comando seguente:

```
$ docker-compose -v
```

L'output dovrebbe mostrare qualcosa di simile:

```
docker compose version 1.16.1, build 6d1ac219
```

Testing del Docker Compose. Il Docker Hub include un'immagine Hello World a puro scopo dimostrativo la quale illustra la configurazione richiesta per eseguire un container con Docker Compose. Creazione di una nuova directory:

```
$ mkdir hello-world
$ cd hello-world
```

Creazione del docker-compose.yaml file:

```
$ $EDITOR docker-compose.yml
```

Di seguito è presente il testo da inserire al suo interno:

```
unixmen-compose-test:
image: hello-world
```

Inserisco da riga di comando la riga seguente allo scopo di avviare il container hello-world.

```
docker-compose up
```

L'output dovrebbe mostrare qualcosa di simile a quando descritto di seguito.

```
Creating helloworld_my-test_1...
Attaching to helloworld_my-test_1
my-test_1 |
my-test_1 | Hello from Docker.
my-test_1 | This message shows that your installation appears to be working
    correctly.
my-test_1 |
```

Verranno ora indicati i file con le relative estensioni necessari allo script per la corretta esecuzione.

- restriction.txt
- docker-compose.yml
- context_level_list.txt

- network.txt

Il primo file /restriction.txt può essere generato ad hoc inserendo per ciascuna riga i campi nome container sorgente, numero container sorgente, nome container destinazione e numero container destinazione, separata dai due punti.

```
nome_container1:numero_container1:nome_container2:numero_container2:
```

Questo file verrà poi interpretato dallo script di automatizzazione con lo scopo di generare le policy e le regole firewall necessarie per ciascun ambiente di container. Tale meccanismo di funzionamento è descritto del manuale programmatore. Il file /docker-compose.yml 5.12 indica le specifiche utili alla creazione di ciascun container. Inoltre è presente un campo *security-opt* il quale permette di definire i campi dell'etichetta di contesto SELinux. In questo caso sono stati definiti il campo *"type"* come svirt_lxc_net_t in modo da limitare l'accesso alle directory e file del sistema host, ai processi in esecuzione all'interno dei container. Inoltre, per poter garantire l'isolamento tra i vari container, sempre in termini di processi che accedono ai file ed alle directory presenti in altri container, è stato definito il campo *"level"* dell'etichetta di contesto.

Il file /context_level_list.txt può essere generato con lo scopo di automatizzare il meccanismo appena descritto. Basterà infatti scrivere all'interno di tale file il valore 0, che dopo essere letto, verrà utilizzato con lo scopo di andare a creare dei range di valori differenti per ciascun campo *"level"* che andrà associato a ciascun container.

L'ultimo file menzionato in precedenza è il file /network.txt il quale è stato utile a definire in modo dinamico una subnet da associare a ciascun ambiente di container. Questo meccanismo è stato utile al fine di poter garantire che ciascun ambiente di container potesse appartenere a delle sottoreti differenti senza che che si andassero a generare delle collisioni di indirizzi ip tra i vari container. Per tale scopo è risultato necessario che l'esecuzione di tutti i container Docker venisse gestita dallo script di automatizzazione yaml_reader.py. Sempre ai fini di una corretta automatizzazione dell'ambiente di container, è stato inoltre sviluppato un semplice script, anch'esso presente nel disco allegato all'interno della directory /Docker_autonomous_clean, con il nome di reset_container.py il cui scopo è stato quello di effettuare un reset delle iptables, di Docker e quindi dei vari container in esecuzione e delle policy SELinux generate per ciascun ambiente di container.

Tutti i file appena descritti sono presenti ed inizializzati al fine di poter effettuare un primo test dello script di automatizzazione della sicurezza all'interno del disco allegato e per comprenderne meglio la struttura in caso di modifiche future. A seguito della corretta installazione e configurazione dell'ambiente di sviluppo si potrà procedere con l'esecuzione dello script. Come esempio si può considerare quello descritto per la tesi, il quale è costituito da un file /docker-compose.yml contenente le specifiche per ciascun container, nonchè le immagini che andranno eseguite all'interno del container. Bisogna precisare che per questo esempio sono state create delle immagini ad hoc. Quindi prima dell'interpretazione del file .yml da parte dello script, risulterà necessario che a priori vengano generate le immagini dei container di *Haproxy* e dell'*Application Service*. L'immagine 5.8 illustra il contenuto della directory principale del CD contenente il software sviluppato.

Per fare ciò bisognerà accedere da terminale all'interno della prima direcotry /haproxy_loadbalancer 5.10 contenuta all'interno della cartella immagini ad-hoc 5.9 per poi inserire il comando seguente:

```
sudo docker build -t haproxy_loadbalancer .
```

Successivamente ci si sposterà all'interno della directory /python_webservice 5.11 per poi inserire sempre da riga di comando la seguente stringa:

```
sudo docker build -t python_webservice .
```

Per la generazione delle immagini utili alla creazione dei container di Haproxy e degli Application Service sarà necessaria la connessione ad internet, in quanto si andranno a scaricare eventuali componenti necessarie non presenti in locale.

In questo modo si sono andate a creare le immagini utili all'esecuzione dell'ambiente di container, le quali sono definite all'interno del /docker-compose.yml file indicato di seguito.

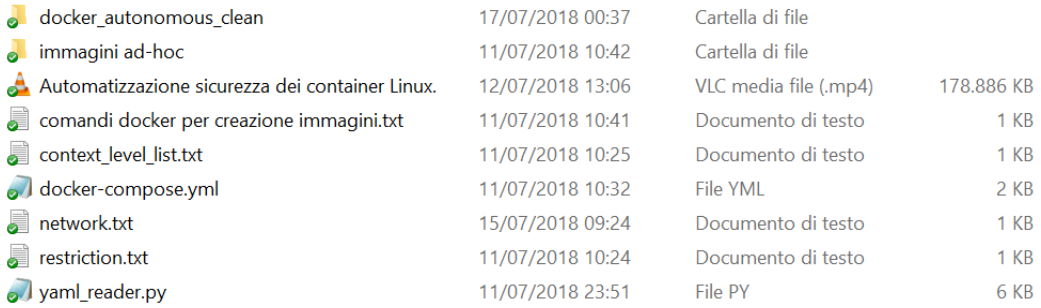

Figura 5.8. Directory principale.

Figura 5.9. Directory contenente le immagini sviluppate ad-hoc.

l'immagine relativa al database mysql verrà scaricata automaticamente da un *repository* sicuro mediante l'utilizzo automatico del comando *docker run* definito all'interno dello script.

Nel caso in cui la procedura di generazione delle immagini sia andata a buon fine, risulterà possibile avviare lo script mediante il comando seguente.

```
sudo python yaml_reader.py
```

Il comando indicato di cui sopra, deve essere eseguito con permessi di root sul sistema Linux in esecuzione all'interno della macchina host, in quanto parte dello script andrà a generare ed applicare delle policy SELinux, dovendo garantire inoltre che SELinux sia settato in modalità *enforcing* al termine della procedura di automatizzazione.

Risulterà inoltre necessario che il file di configurazione /docker-compose.yml sia presente nella stessa directory in cui è presente il software di automatizzazione.

Dopo l'avvio dell'ambiente di container risulterà possibile effettuare un test accedendo in localhost dal browser, in modo tale da poter visionare il comportamento del container *Haproxy* il quale si occuperà di bilanciare le richieste sui due *Application service* mediante l'algoritmo di *Round Robin*. Adesso sarà possibile accedere mediante comando *docker exec* dal container di *Haproxy* con lo scopo di eseguire un *ping* verso il container *Mysql*. In questo modo sarà possibile vedere come il traffico di tipo ICMP venga bloccato tra i due container *Haproxy* e *Mysql*.

Un esempio di funzionamento di tale script è stato mostrato in una demo video [68] sviluppata appositamente per il caso di studio preso in esame in questa tesi. Tale dimostrazione è presente nel CD che è stato consegnato alla commissione di tesi.

```
[alias@localhost haproxy_loadbalancer]$ sudo docker build -t haproxy_loadbalancer .
Sending build context to Docker daemon    428MB
Step 1/2 : FROM haproxy:1.7
 ---> 4788186a51c3
Step 2/2 : COPY /etc/haproxy/haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
 ---> Using cache
 ---> fe4bf82eac0c
Successfully built fe4bf82eac0c
Successfully tagged haproxy_loadbalancer:latest
[alias@localhost haproxy_loadbalancer]$
```

Figura 5.10. Creazione immagine haproxy_loadbalancer.

```
[alias@localhost python_webservice]$ sudo docker build -t python_webservice .
Sending build context to Docker daemon  7.168kB
Step 1/9 : FROM python:2
2: Pulling from library/python
0bd44ff9c2cf: Pull complete
047670ddbd2a: Pull complete
ea7d5dc89438: Pull complete
ae7ad5906a75: Pull complete
0f2ddfdfc7d1: Download complete
85124268af27: Download complete
1be236abd831: Download complete
fe14cb9cb76d: Download complete
cb05686b397d: Download complete
```

Figura 5.11. Creazione immagine python_webservice.

```
version: "3"

services:

    db:
      image: mysql
      container_name: mysql-server
      ports:
            - 3306
      security_opt:
        - label:type:svirt_lxc_net_t
        - label:level:s0:c0.c143
      environment:
       MYSQL_ROOT_PASSWORD: secret

    web1:
        image: python_webservice
        container_name: web1
        ports:
            - 8085:80
        security_opt:
        - label:type:svirt_lxc_net_t
        - label:level:s0:c144.c287

    web2:
        image: python_webservice
        container_name: web2
        ports:
            - 8086:80
        security_opt:
        - label:type:svirt_lxc_net_t
        - label:level:s0:c288.c431

    haproxy:
            image: haproxy_loadbalancer
            volumes:
            - /haproxy/etc/haproxy:/usr/local/etc/haproxy:ro
            container_name: haproxy
            ports:
            - 80:80
            security_opt:
            - label:type:svirt_lxc_net_t
            - label:level:s0:c432.c575
```

Figura 5.12. Docker-compose.yml file.

Capitolo 6

Risultati

Durante le prove implementative eseguite allo scopo di automatizzare la sicurezza tra i container, oltre all'isolamento degli stessi dal sistema host, mediante applicazione di policy SELinux, il quale inizialmente ha portato a dei problemi relativi alla comunicazione di rete tra più container docker appartenenti alla stessa rete Docker. Infatti anche dopo l'assegnazione delle regole all'interno delle *iptables*, la limitazione dell'inoltro dei pacchetti tra due specifici container, non andava a buon fine.

Infatti all'atto di tale assegnazione, è stato eseguito l'accesso alla bash di entrambi i container, quali *Haproxy* e *Mysql*, allo scopo di poter testare l'effettivo funzionamento delle regole precedentemente imposte, le quali dovevano impedire l'inoltro dei pacchetti tra i due container, mediante un semplice test che consiste nell'utilizzo del comando *ping* seguito dall'indirizzo IP destinazione. I pacchetti ICMP venivano inoltrati correttamente.

Questo problema è sorto perchè Docker di default, permette l'*Inter Communication Container* per ciascuna rete Docker. Questo permetteva ai container di comunicare di default tra loro, andando quindi a bypassare anche le regole di inoltro dei pacchetti definite all'interno delle iptables. E' risultato quindi necessario disabilitare tale funzionalità, all'atto della creazione della rete Docker che sarebbe dovuta essere assegnata allo specifico ambiente di container, attraverso l'utilizzo del comando indicato di seguito.

```
com.docker.network.bridge.enable_icc=false
```

Tale comando deve essere inserito alla creazione della rete Docker come segue.

```
sudo docker network create --subnet=172.18.0.0/16 --opt
    com.docker.network.bridge.enable_icc=false env_net
```

Bisogna precisare inoltre, che la disabilitazione di tale funzionalità, non limita la comunicazione tra i container, infatti questi posso sempre comunicare tra di loro in modo corretto, ma non va ad impedire che la iptables gestisca la comunicazione di rete tra di essi.

Di seguito vengono illustrati i risultati ottenuti con la policy SELinux abilita. Come evidenziato nella figura 6.1, SELinux è in modalità di enforcing, quindi sta applicando tutte le policy definite e abilitate nel sistema host. La lista completa delle policy attive presenti nel sistema viene richiamata attraverso l'utilizzo del comando semodule -l. Una parte di questa lista è mostrata in figura dove viene evidenziata la policy secmark 1.0, che è stata utilizzata per tale scopo, ovvero, questa policy viene richiamata all'atto della definizione di una regola che viene assegnata all'interno delle iptables del sistema host, come illustrato in testa alla figura 6.1. Nella parte centrale sinustra di tale figura 6.1 sono illustrati i risultati dei ping effettuati accedendo in bash tramite il comando docker exec al container haproxy e al container mysql-server. Non è stato quindi possibile effettuare il ping tra i due container a causa della policy SELinux precedentemente definita e richiamata tramite regole firewall. E' stato invece possibile comunicare tramite l'haproxy container attraverso il browser con i due Application Service identificati con ID: be31b63d94a0 e ID: a0da3f90235a rispettivamente.

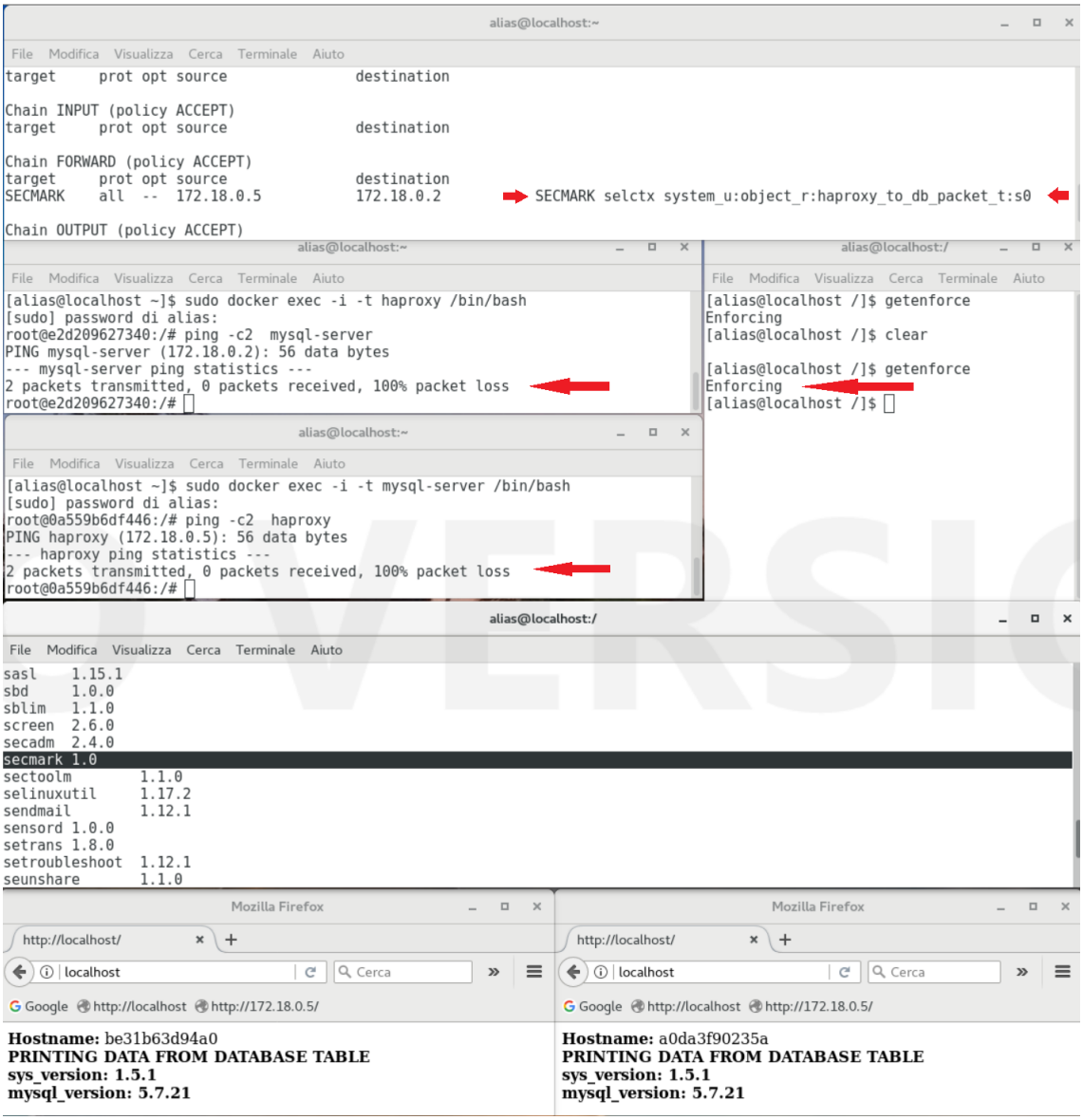

Figura 6.1. Risultati ottenuti con SELinux abilitato in modalità enforcing.

Disabilitando SELinux, ovvero settandolo in modalità permissive, è stato possibile ottenere i seguenti risultati6.2. Anche in questo caso, il container haproxy riesce a comunicare correttamente con i due application services, i quali restituiscono i risultati acquisiti durante la lettura di un database comune in esecuzione sul container mysql-server. E' anche possibile vedere, come la comunicazione diretta tra il container di haproxy e mysql-server, vada a buon fine, a causa della policy SELinux disabilitata 6.2.

Un ulteriore prova è stata effettuata mantenendo SELinux abilitato, ovvero settato in modalità di enforcing e rimuovendo dalla lista la policy secmark utilizzata per limitare la comunicazione tra i due container haproxy e mysql-server6.3. Anche in questo caso, pur essendo SELinux abilitato, la comunicazione tra i due container va a buon fine, nonostante la regola assegnata all'interno della iptables risulti ancora definita correttamente.6.3. Tale regola infatti permette la gestione del traffico basandosi sulla policy definita al suo interno. In assenza di tale policy, tale regola non avrà alcun significato, ma rimarrà sempre memorizzata all'interno della iptables.

Nella figura 6.4 è illustrata la definizione del modulo di policy Secmark. Oltre alla definizione di nome e versione modulo, le righe successive presentano, viene richiamato l'etichetta di contesto svirt_lxc_net_t attraverso la funzione genrquire(...). Questo passaggio è utile a gestire i pacchetti

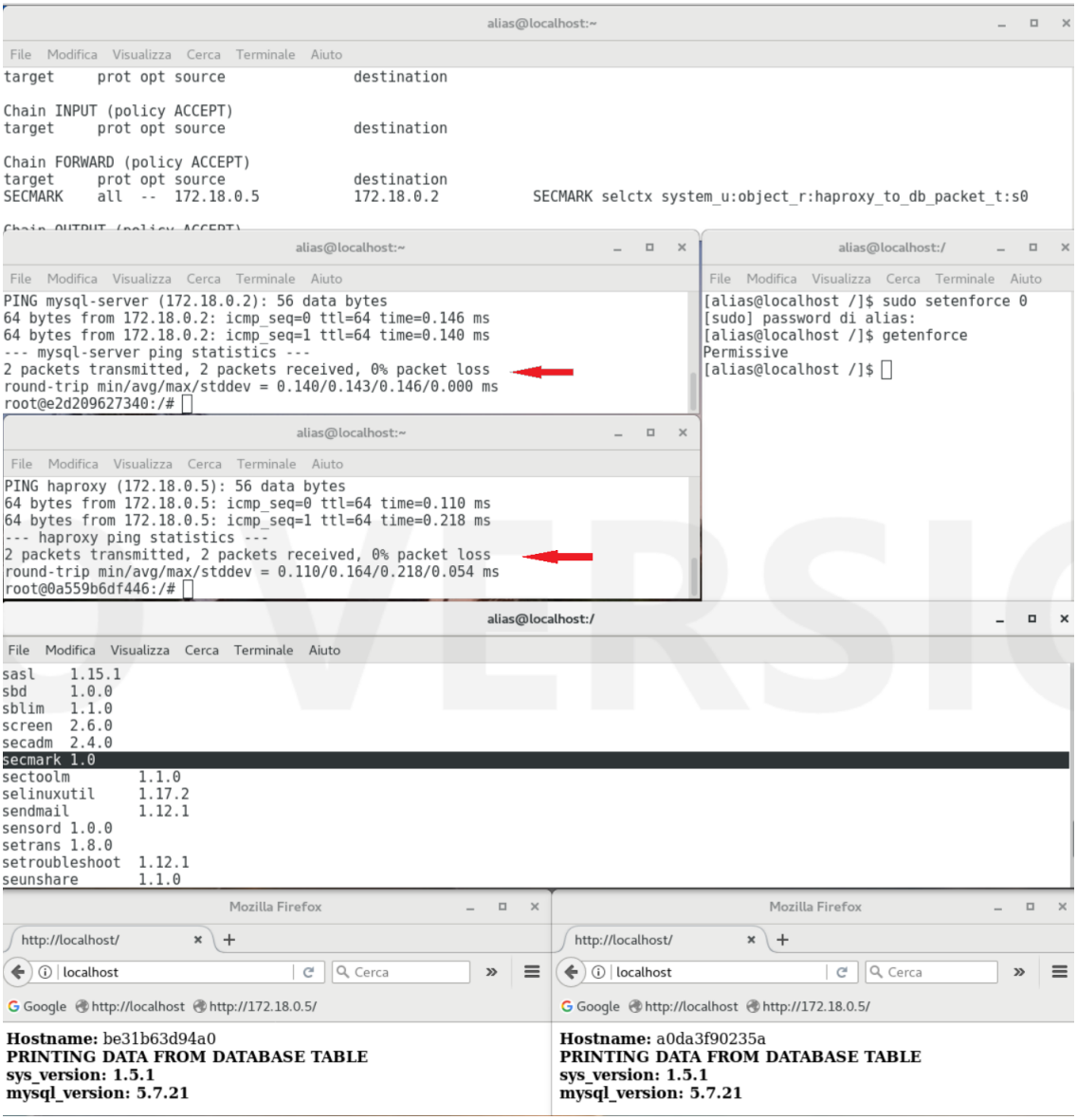

Figura 6.2. Risultati ottenuti con SELinux disabiltato.

instradati da processi la cui etichetta di contesto e quella indicata di cui sopra, ovvero i processi in esecuzione all'interno dei container Docker. Attraverso l'utilizzo del comando:

```
docker inspect container_name
```

è possibile visualizzare l'etichetta che ad esempio è stata assegnata al container haproxy, come illustrato in figura 6.5. Queste etichette sono state definite allo scopo di poter gestire il traffico ciascun container. Infatti per poter gestire la comunicazione e quindi l'inoltro e la ricezione dei pacchetti tra i vari container, è necessario conoscere sia l'etichetta dei container in questione, sia l'etichetta assegnata ai pacchetti in uscita da una specifica sorgente, che in questo caso le nostre sorgenti saranno il container haproxy e il mysql-server. le righe di dontaudit all'interno del modulo di policy definiscono le seguenti regole:

A tutti i pacchetti etichettati come aproxy_to_db_packet _t, il cui container è etichettato come svirt_lxc_t viene negato l'inoltro. Da notare che vengono negati solo i pacchetti etichettati come aproxy_to_db_packet _t ovvero quelli in uscita dall'haproxy container e diretti verso il mysql-server container e non i pacchetti diretti verso altre destinazioni 6.4.

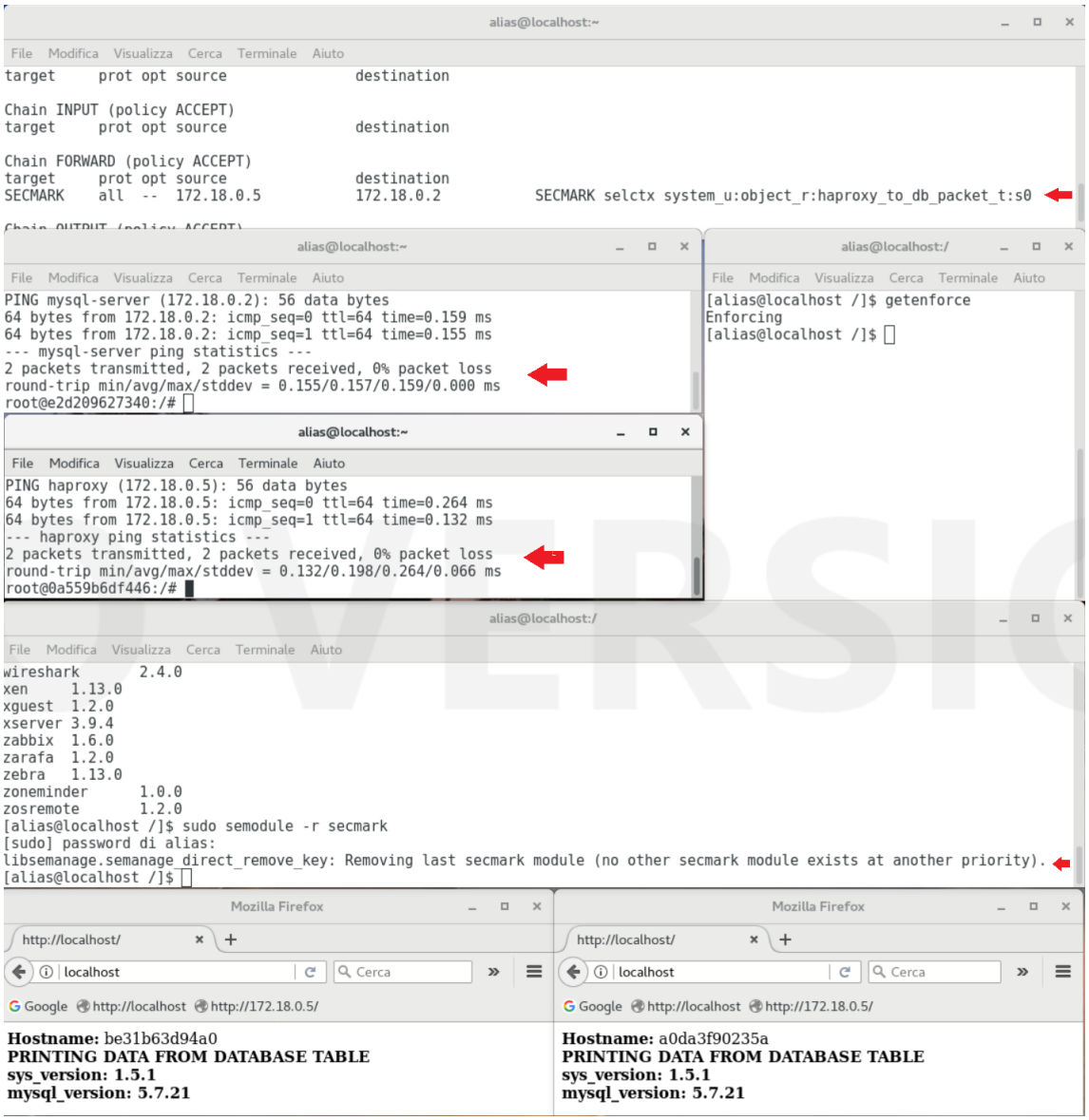

Figura 6.3. Risultati ottenuti con policy SELinux rimossa dalla lista.

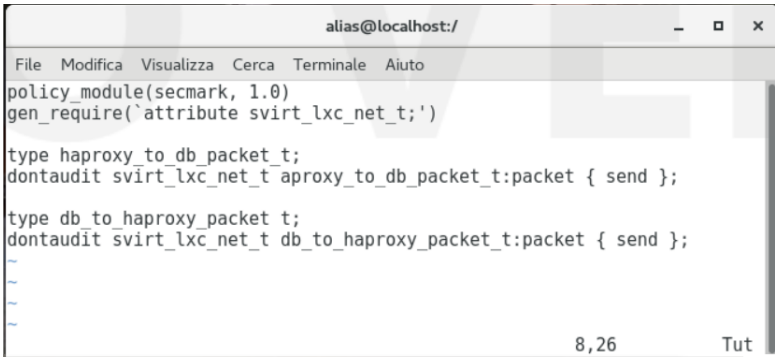

Figura 6.4. Policy module Secmark 1.0.

La riga successiva invece,indica che a tutti i pacchetti etichettati come db_to_haproxy_packet _t, il cui container è etichettato come svirt_lxc_t viene negato l'inoltro. Anche in questo caso,

vengono negati solo i pacchetti etichettati come db_to_haproxy_packet _t ovvero quelli in uscita dal mysql-server container e diretti verso l'haproxy container e non i pacchetti diretti verso altre destinazioni 6.4.

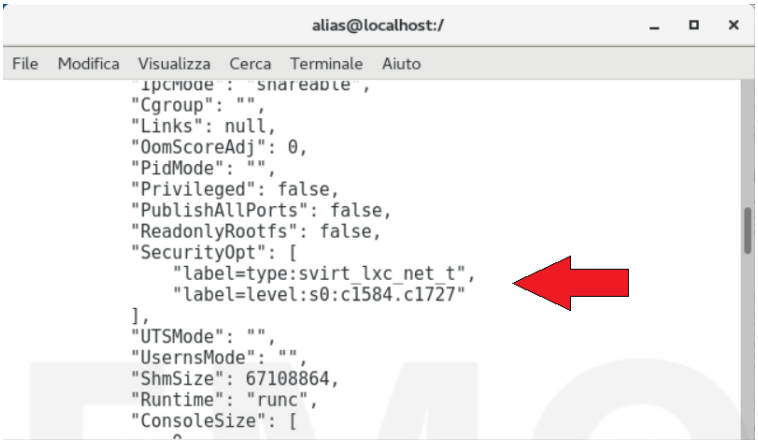

Figura 6.5. Utilizzo del comando docker inspect sul container hproxy.

Con questa nuova soluzione di sicurezza risulta importante evidenziare come alcuni tipi di attacchi possono essere mitigati rispetto alle soluzioni native proposte dalle funzionalità di Docker. Infatti, come descritto nell'analisi presente nella sezione 5 dell'articolo [60], Docker fornisce un buon livello di isolamento e di limitazione delle risorse assegnabili a ciascun container, attraverso l'utilizzo dei *namespaces* o *filesystem namespaces* (*Filesystem Isolation*) [60] [61], *Cgorups* [61], già nella sua configurazione di default. Un problema che però è stato riscontrato, è correlato al modello di rete di default. Infatti, il network bridge di default risulta essere vulnerabile ad attacchi di ARP spoofing [63] e MAC flooding [69], dato che Docker non fornisce alcun filtro sul traffico di rete. Possibili soluzioni riscontrate negli anni precedenti, sono l'utilizzo di ebtable [62] o quello delle reti virtuali [70]. Attraverso l'utilizzo di regole firewall in combinazione alle policy SELinux, risulta quindi possibile mitigare attacchi di spoofing. Invece, l'assegnazione di etichette SELinux a file e directory dedicate a ciascun container, permette di evitare che un utente malevolo possa alterare il file di configurazione di un reverse proxy che agisce come load-balancer verso un numero definito di servizi web i quali accedono ad una risorsa condivisa.

Capitolo 7

Conclusioni e sviluppi futuri

In questo capitolo si andrà a descrivere in modo sintetico, un possibile sviluppo futuro relativo alla parte implementativa da me sviluppata, allo scopo di automatizzare la sicurezza dei container in ambiente Linux in combinazione con l'automatizzazione di policy SElinux e regole firewall. Una possibile soluzione futura potrebbe basarsi sul migliorare lo script di automazione della sicurezza sviluppato e descritto in questa tesi, con lo scopo di andare a gestire applicazioni in esecuzione sempre all'interno di container, ma su macchine differenti e non più sullo stesso host. L'idea potrebbe essere quella di modificare lo script in modo tale che questo possa essere eseguito in modo automatico da molteplici macchine connesse attraverso una medesima rete. L'evoluzione di tale algoritmo, di base dovrebbe come sempre andare a leggere un file manifest di configurazione il quale al suo interno indicherà quali applicazioni dovranno essere eseguite sulla macchina n, all'interno di quali container, quali risorse dovranno essere assegnate ed in che quantità, che tipo di policy di sicurezza SElinux si dovranno andare a generare su tale macchina, e poichè ci si andrà a basare sullo strumento Labeled-IPsec, andrà definito inoltre quale livello di cifratura dovranno avere le SA generate per instaurare una connessione sicura tra i diversi sistemi, in modo tale da poter prediligere se necessario, le prestazioni a discapito del livello di Encryption fornito da IPsec 3.1.10. Poichè la configurazione di ogni sistema possa andare a buon fine, risulterà quindi necessario che tutte le macchine abbiano il supporto a SELinux e che questo sia a sua volta abilitato, inoltre lo script di automatizzazione dovrà appartenere ad un dominio SElinux ben specifico e uguale per ogni macchina. Questo dominio fornirà permessi elevati alle applicazioni che verranno eseguite al suo interno. Per questo motivo è molto importante che venga sempre garantita l'integrità e l'autenticità di tale script, allo scopo di evitare che un programma con tali permessi, se modificato da un utente malevolo, possa andare a danneggiare in modo virale tutti i sistemi connessi alla medesima rete. L'automatizzazione dello script a livello multi-host, in associazione al meccanismo di sicurezza Labeled-IPsec, porterebbe al mitigare possibili attacchi *Ransomware*, i quali generalmente sono sviluppati con lo scopo di cifrare le informazioni di una generica macchina host sotto attacco con chiavi di cifratura complesse a 2048 o 4096 bit in modo tale da renderne impossibile o quasi la decifratura delle informazioni. Un recente attacco di questo genere è stato portato ad un'azienda presente in Italia, la *Nuance* la quale ha filiali presenti in varie parti del mondo e che si occupa dello sviluppo di software per sintesi vocale. Infatti per tale attacco è stato utilizzato il Petya Ransomware, il quale a differenza degli altri Ransomware si estende in modo virale andando ad "infettare " non solo l'intero sistema host sotto attacco, ma anche tutti i dispositivi connessi alla medesima rete o connessi in remoto attraverso una rete VPN.

> "The NotPetya attacks were different from the previous Petya ransomware strain in that it was modified to include worm functionality. Petya ransomware took advantage of unpatched and outdated systems, and would encrypt the master boot records of infected Windows computers, according to US-CERT.
>
> The Petya variant is a self-propagating worm that can laterally move through an infected network by harvesting credentials and active sessions on the network, exploiting previously identified SMB vulnerabilities, and using legitimate tools such as the Windows Management

> Instrumentation Command-line (WMIC) tool and the PsExec network management tool, Industrial Control Systems (ICS)-CERT stated on its website.
>
> An affected system will then scan the local network for additional systems to infect. The strain will then encrypt files and overwrite the Master Boot Record or wipe parts of the disk drive." [71].

Infatti con la soluzione menzionata in precedenza, attraverso domini SELinux e l'utilizzo del tool di sicurezza Labeled-IPsec, sarebbe stato possibile mitigare tale tipo di attacco. Sarebbe stato però necessario l'utilizzo di macchine Linux e non Windows, in quanto tali strumenti di sicurezza sono disponibili solo per sistemi Linux. In questo modo si sarebbero andati a creare due domini SELinux ben distinti, uno locale ed uno remoto associati a 2 differenti coppie di *Security Association*(SA) per IPsec 3.1.10, sempre una locale ed una remota. Attraverso tale configurazione, le applicazioni software in arrivo dalla SA remota, si sarebbero potute "muovere " solo all'interno del dominio remoto identificato da SELinux, e quindi nel caso di un attacco malevolo sarebbero stati infettati solo i file e le directory appartenenti a tale dominio, ma non la parte del disco di ogni macchina host sotto attacco, dedicata al dominio SELinux di tipo locale. Le applicazioni necessarie per le attività lavorative verrebbero invece memorizzate in sistemi server appartenenti al dominio SELinux di tipo locale e configurati con coppie di SA di tipo locale e non remoto. Allo stesso modo, anche la parte relativa alla rete VPN aziendale, apparterrà ad un dominio SELinux locale con relative SA locali. In questo modo la rete interna aziendale che instaura la connessione virtuale tra le varie sedi presenti a livello mondiale, non sarà affetta in alcun modo da software proveniente da internet, ma quest'ultimo sarà di libero utilizzo solo su quella parte di disco di ciascun sistema host etichettata come "remota " attraverso policy SELinux.

Bibliografia

[1] NCC Group: Understanding and Hardening Linux Containers,
https://www.nccgroup.trust
[2] User Experience
https://en.wikipedia.org/wiki/User_experience
[3] About containerd
https://containerd.io/
[4] What is CRUD?
https://www.codecademy.com/articles/what-is-crud
[5] Docker:introducing execution drivers and libcontainer
https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/
[6] Libvirt
https://libvirt.org/
[7] The container networking
https://thenewstack.io/container-networking-landscape-cni-coreos-cnm-docker/
[8] Container technologies: more than just Docker
https://cloudacademy.com/blog/container-technologies-more-than-dockers/
https://cloudacademy.com/blog/container-technologies-more-than-dockers/
[9] Host Rebinding Attack
https://www.blackhat.com
[10] LXC containers in Ubuntu Server 14.04 LTS
http://en.community.dell.com
[11] Operating System Containers vs. Application Containers
https://blog.risingstack.com
[12] Docker Scheduler
http://thingsoncloud.com/2015/03/docker-scheduler/
[13] LibContainer Overview
http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/
[14] LibContainer Overview
http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/
[15] The Containers Ecosystem
https://www.slideshare.net
[16] Docker 1.11 et plus: Engine is now built on runC and containerd
https://medium.com
[17] CRIU: Time and Space travel Service for Linux Applications
https://www.slideshare.net
[18] Realizing Linux Containers
https://www.slideshare.net
[19] What is libnetwork?
https://blog.docker.com
[20] The Container Networking Landscape: CNI from CoreOS and CNM from Docker
https://thenewstack.io
[21] LXC vs Docker Comparison
https://robinsystems.com
[22] LXC vs Docker Comparison
https://www.packtpub.com

[23] Docker vs Rkt
https://bobcares.com/blog/docker-vs-rkt-rocket/
[24] tcp-dns
https://www.slideshare.net/culvertonblessy/12-tcpdns
[25] http://onecloudclass.com/lab-1-introduction/
http://onecloudclass.com/lab-1-introduction/
[26] Introducing Docker Content Trust
https://blog.docker.com/2015/08/content-trust-docker-1-8/
[27] Seccomp Archiecture
https://www.researchgate.net
[28] SELinux Workshop
https://www.slideshare.net/Jhon_Masschelein/se-linux-42322789
[29] Using SELinux and iptables Together
https://www.linux.com/learn/using-selinux-and-iptables-together
[30] New secmark-based network controls for SELinux
http://james-morris.livejournal.com/11010.html
[31] SELinux Networking Support
http://selinuxproject.org/page/NB_Networking
[32] LSM, SELinux, Netlabel, and CIPSO
http://grantcurell.com/2017/03/10/lsm-selinux-netlabel-and-cipso/
[33] Administration of Labeled IPsec
https://docs.oracle.com/cd/E26502_01/html/E29017/txnet-20.html
[34] Network Labeling Statements
https://selinuxproject.org/page/NetworkStatements
[35] Docker Host Networking Modes
http://www.abusedbits.com
[36] Sysfs and Procfs
https://www.missinglinkelectronics.com
[37] Let There Be Light Sysdig Adds Container Visibility
https://sysdig.com
[38] ContainerCon sysdig Slides
https://www.slideshare.net
[39] How to Secure Containers
https://www.slideshare.net/Sysdig/how-to-secure-containers
[40] A Look Back at One Year of Docker Security
https://blog.docker.com/2016/04/docker-security/
[41] docker-bench-security: Audit Well-Known Docker Vulnerabilities
https://www.dennyzhang.com/docker_bench_security
[42] Docker container networking
https://docs.docker.com/engine/userguide/networking/
[43] What is Docker?
https://indrabasak.wordpress.com/2017/04/03/what-is-docker/
[44] Install and Setup LXC Linux Containers on CentOS/RHEL/Ubuntu
http://www.hackthesec.co
[45] Isolation with Linux Containers
https://deis.com/blog/2015/isolation-linux-containers/
[46] Isolation with Linux Containers
https://csrc.nist.gov/projects/role-based-access-control
[47] Yama - The Linux Kernel Archives
https://www.kernel.org/doc/Documentation/security/Yama.txt
[48] CVE - Common Vulnerabilities and Exposures (CVE)
https://www.cvedetails.com/
[49] Exploit Database
https://www.exploit-db.com/
[50] Use volumes — Docker Documentation
https://docs.docker.com/engine/admin/volumes/volumes/

[51] NetLabel
www.paul-moore.com/docs/netlabel-linuxcon-09212009.pdf
[52] SELinux Network Control
www.paul-moore.com/docs/selinux_network_controls-pmoore-122012-r1.pdf
[53] IPSec
https://ccie-study.wikispaces.com/IPSec
[54] Secure Networking with SELinux
https://securityblog.org/2007/05/28/secure-networking-with-selinux/
[55] Setkey
https://www.systutorials.com/docs/linux/man/8-setkey/
[56] Introducing Docker Content Trust
https://blog.docker.com/2015/08/content-trust-docker-1-8/
[57] Docker run
https://docs.docker.com/engine/reference/commandline/run/#options
[58] Docker compose
https://docs.docker.com/compose/overview/
[59] Dockerfile
https://docs.docker.com/engine/reference/builder/
[60] Analysis of Docker Security
https://arxiv.org/pdf/1501.02967
[61] Docker security
https://docs.docker.com/engine/security/security/
[62] ebtables
http://ebtables.netfilter.org/
[63] ARP Spoofing
https://www.veracode.com/security/arp-spoofing
[64] How to Install CentOS 7 Step by Step
https://linoxide.com/how-tos/centos-7-step-by-step-screenshots/
[65] Get Docker for CentOS
https://docs.docker.com/install/linux/docker-ce/centos/
[66] An Introduction to SELinux on CentOS 7
https://www.digitalocean.com/community/tutorials/an-introduction-to-selinux-on-centos-7-part-1-basic-concepts
[67] How to Install Python 3.6.4 on CentOS 7
https://www.rosehosting.com/blog/how-to-install-python-3-6-4-on-centos-7/
[68] Demo video software di automatizzazione della sicurezza per i container Linux
https://db.tt/a4uBMBPy7e
[69] MAC flooding
http://www.infosecisland.com/blogview/3684-How-to-Detect-a-Mac-Flooding-Attack.html
[70] Virtual network interface
https://en.wikipedia.org/wiki/Virtual_network_interface
[71] Nuance Restores Clients After NotPetya Malware Attack
https://healthitsecurity.com/news/nuance-restores-75-of-clients-after-notpetya-malware-attack