POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Progettazione di funzioni di sicurezza di rete virtuali

Relatori
prof. Antonio Lioy
dott. Marco De Benedictis

Candidato
Vincenzo Paolo Bacco

Anno accademico 2016-2017

Sommario

La continua evoluzione delle tecniche di attacco rivolte a sistemi informatici costringe utenti e organizzazioni ad un continuo aggiornamento dei propri meccanismi di difesa. Il crescente numero di dispositivi connessi ad Internet richiede l'impiego di strumenti sempre più rapidi nel contrastare qualsiasi forma di minaccia, sia in ambito locale sia Cloud.

Il presente elaborato discute una tecnologia per la difesa dei sistemi informatici basata su un'infrastruttura NFV. Esso espone il concetto di *virtual Network Security Functions* (vNSFs), ossia VNF orientate alla sicurezza, e lo contestualizza in uno scenario di virtualizzazione basata su container. Inoltre, esamina il caso di studio di una vNSF avente funzioni di Reverse Proxy e WAF, denominata vNSF Reverse Proxy.

La trattazione parte da un'analisi dei principali aspetti di sicurezza legati all'uso di container in un ambiente di produzione, introducendo le potenziali minacce e le principali best-practice nella configurazione.

Successivamente, valuta diversi software open-source, sia nell'ambito dei Reverse Proxy sia in quello dei WAF. Dai test e dalle relative considerazioni emerge come la soluzione più idonea al contesto vNSF sia costituita da httpd e ModSecurity.

In seguito, si introduce una possibile architettura di vNSF basate su container, trattando allo stesso tempo sia aspetti di carattere generale sia specifici per la vNSF Reverse Proxy. Si evidenzia la presenza di convergenze e divergenze tra l'architettura proposta e l'attuale stato della standardizzazione ETSI su NFV.

Inoltre, si propone una possibile implementazione della vNSF Reverse Proxy con la piattaforma Docker. Essa include, in aggiunta ad httpd e ModSecurity, componenti ausiliari al funzionamento sviluppati nel contesto del presente lavoro di tesi. A tal proposito, è stato utilizzato il linguaggio di programmazione Go.

Infine, si presentano i risultati derivanti dal collaudo della vNSF Reverse Proxy e le relative conclusioni, evidenziando i punti di forza e i punti di debolezza della soluzione proposta. In particolare, si evince come la configurazione di una vNSF debba essere effettuata considerando attentamente l'ambiente di produzione in cui questa è adoperata.

Indice

Capitolo 1

Introduzione

Nella pubblicazione *The NIST Definition of Cloud Computing*, il *National Institute of Standards and Technology* (NIST) definisce il *Cloud Computing* come un modello computazionale che prevede l'accesso remoto, la configurazione e l'utilizzo di un insieme di risorse con un intervento minimo o nullo da parte di un fornitore, detto *provider* [1]. Nel corso dell'ultimo decennio, il termine *Cloud* si è diffuso in diversi contesti. Il suo utilizzo varia da scenari scientifici ad ambienti amministrativi, rivelandosi come tecnologia dominante ed estremamente versatile.

Nell'ambito della stessa pubblicazione, il NIST definisce tre principali modelli di servizio basati su Cloud Computing: *Software as a Service* (SaaS), *Platform as a Service* (PaaS) e *Infrastructure as a Service* (IaaS). Ogni modello fornisce un diverso livello di astrazione e un diverso tipo di servizio.

Il paradigma SaaS prevede che l'utente utilizzi applicazioni offerte sull'infrastruttura hardware e software del provider. Egli accede a tali servizi mediante apposite interfacce ed ha la possibilità di configurare aspetti relativi esclusivamente la sua utenza, ossia il suo *account*. Qualsiasi sua impostazione non ha effetto sugli altri utenti dell'applicazione.

Il modello PaaS consente all'utilizzatore di sfruttare l'infrastruttura del provider per l'esecuzione di applicazioni proprie o di terzi. In questo caso, egli ha possibilità di modificare il funzionamento e la configurazione dell'applicazione. Tuttavia, non può intervenire sull'architettura software e hardware ad essa sottostante.

Il paradigma IaaS prevede che l'utente possa gestire autonomamente la configurazione di diverse risorse offerte, quali Sistemi Operativi, reti e dischi. Tuttavia, egli non ha visibilità sull'architettura hardware e software alla base dell'infrastruttura. Per ovvi motivi, la gestione di quest'ultima è riservata al provider.

Nel tempo sono stati introdotti ulteriori paradigmi, i quali specializzano i succitati fornendo nuovi tipi di servizio es. *Security as a Service* (SecaaS), *Blockchain as a Service* (BaaS), *Mobile Backend as a Service* (MBaaS).

Dal punto di vista tecnologico, il Cloud Computing affonda le sue radici nel concetto di *virtualizzazione*. La virtualizzazione consente di superare la già obsoleta accezione di sistema informatico come entità fisica, consentendo la creazione di rappresentazioni software per dispositivi hardware. In altri termini, permette di emulare componenti non concretamente esistenti rivoluzionando i modelli computazionali utilizzati in passato. Un esempio basilare del concetto di virtualizzazione è la possibilità di eseguire più sistemi operativi contemporaneamente all'interno della stesso server fisico. Tra le numerose tecnologie correlate al concetto di virtualizzazione vi è l'architettura *Network Functions Virtualization* (NFV).

NFV è un paradigma che si prefigge l'obiettivo di virtualizzare le apparecchiature adibite al networking es. router, bridge, eseguendole come software all'interno di server fisici [2, Overview]. Questi ultimi possono essere localizzati allo stesso tempo in grossi Data Center o piccole infrastrutture. Le implementazioni software di dispositivi fisici che svolgono funzioni di networking, all'interno del mondo NFV, sono denominate *Virtual Network Function* (VNF).

Il presente lavoro di tesi si colloca in uno scenario fondato su tre aspetti: l'analisi e lo sviluppo di VNF adibite a funzioni di sicurezza dette *virtual Network Security Function* (vNSF), l'impiego di un determinato tipo di virtualizzazione, detto *virtualizzazione leggera*, e la contestualizzazione all'interno del paradigma SecaaS.

Il paradigma SecaaS amplia la portata dei modelli succitati introducendo la sicurezza, *da remoto*, dei sistemi informatici. Si concretizza nella fornitura, da parte di un provider, di funzioni di sicurezza come servizi Cloud. Tra queste rientrano, ad esempio, autenticazione, anti-malware e intrusion detection.

Una vNSF è un tipo di VNF utilizzata per implementare il paradigma SecaaS all'interno dell'architettura NFV. Essa può limitarsi a monitorare una data infrastruttura o intervenire direttamente [2, Security as a Service]. Nel primo caso può essere utilizzata come *honeypot*, ossia un'esca per studiare il comportamento di un attaccante che tenta di accedere illegittimamente a dati di valore. Nel secondo caso può essere utilizzata per prevenire o fermare un attacco. Ad esempio, può essere adoperata come anti-malware all'interno di una rete aziendale.

La virtualizzazione leggera, come si vedrà nella trattazione successiva, è una tecnologia innovativa che supera il concetto di applicazione come blocco monolitico, predisposto a svolgere un vasto numero di operazioni, con un modello più modulare detto a *microservizi*. Questo propone un'elaborazione maggiormente distribuita, dove non vi è più un singolo software che svolge diverse funzioni, ma diversi software tra loro interconnessi che eseguono ciascuno una singola funzione.

Il presente elaborato si pone l'obiettivo di analizzare, progettare e implementare un *Proof of Concept* (PoC), ossia un prototipo, di vNSF mediante virtualizzazione leggera. Si è scelto di implementare un caso d'uso specifico, rappresentato da un Reverse Proxy con capacità di filtraggio, detto vNSF Reverse Proxy.

La trattazione si articola in diverse fasi. Nel Capitolo 2, si esaminano i principi di virtualizzazione leggera, confrontandola con altre tecnologie presenti nello scenario attuale e valutandone gli aspetti di sicurezza. In seguito, il Capitolo 3 introduce il concetto di Reverse Proxy, elencando le principali implementazioni del mondo open-source. Dato che la vNSF Reverse Proxy si predispone sia a funzioni di proxy sia di filtraggio, nello stesso momento si presenta la figura del *Web Application Firewall* (WAF), esaminando anche qui le principali soluzioni presenti sul mercato. Successivamente, nel Capitolo 4 e nel Capitolo 5, si procede rispettivamente alla progettazione e all'implementazione della vNSF Reverse Proxy, cercando, tuttavia, di delineare anche aspetti architetturali generali, utilizzabili in futuro per altri tipi di vNSF. Infine, il Capitolo 7 presenta i risultati del collaudo dell'attuale implementazione della vNSF Reverse Proxy in diversi scenari.

Capitolo 2

Analisi di sicurezza dei container

2.1 Fondamenti di virtualizzazione

Il Cloud Computing vede tra i principi generali il concetto di virtualizzazione. Questo indica la possibilità di astrarre risorse hardware creandone delle rappresentazioni software. È stimato che la maggior parte dei server impieghi solo il 12% delle potenzialità dei componenti hardware a disposizione [3, pag. 3]. L'utilizzo della virtualizzazione consente di incrementare tale percentuale mediante una ripartizione più efficiente dell'architettura fisica dei server.

Alla base della virtualizzazione ci sono i concetti di *host* e *guest*. Un host è un sistema che può accedere direttamente ai dispositivi fisici presenti nell'architettura di un sistema di calcolo, ad esempio un server. Al contrario, un guest è un'entità, eseguita nel contesto di un sistema host, che si interfaccia o condivide funzionalità con il sistema host per poter accedere a risorse hardware.

È possibile suddividere le tecnologie di virtualizzazione in due categorie: *hardware* o *leggera*.

2.1.1 Virtualizzazione hardware

La virtualizzazione hardware, denominata anche virtualizzazione basata su *hypervisor*, prevede l'impiego di *Virtual Machine* (VM). Una VM è un'istanza di un Sistema Operativo in esecuzione all'interno di un ambiente virtualizzato. Essa è un sistema guest costituito da un insieme di processi eseguiti nel contesto di un sistema host [4].

Una VM riproduce un Sistema Operativo con tutte le sue funzionalità, a partire da quelle per l'utente fino a quelle per gli sviluppatori. In particolare, una VM possiede un proprio *kernel*.

Il kernel è lo strato software del Sistema Operativo che si occupa dell'interazione diretta e il controllo dell'hardware presente in un calcolatore. Esso contiene tutte le funzionalità e interfacce che consentono l'esecuzione di qualsiasi applicativo per l'utente.

Una VM è completamente isolata e indipendente da altri processi in esecuzione su un host. Tuttavia, sebbene contenga al suo interno un Sistema Operativo, essa dispone solo di una rappresentazione software (o virtualizzata) delle risorse hardware a sua disposizione. Pertanto, il kernel di una VM si interfaccia tipicamente con dispositivi emulati. Ad esempio, esso controlla una *virtual CPU* (vCPU) e una *virtual RAM* (vRAM).

La corrispondenza di vCPU e vRAM con la CPU e la RAM dell'host avviene mediante un ulteriore strato software denominato hypervisor. Esso permette l'interazione tra i dispositivi virtuali visti da una VM con l'hardware reale presente nel server.

Analogamente ad un Sistema Operativo in un host, una VM possiede un numero predefinito, non modificabile durante la sua esecuzione, di risorse computazionali come vCPU e vRAM. Un core in una vCPU corrisponde al più ad un core nella CPU e la dimensione di una *pagina* (blocco) di memoria vRAM non eccede quella di una pagina in RAM.

2.1.2 Virtualizzazione leggera

La virtualizzazione leggera si basa sul concetto di container. Esso rappresenta un ambiente isolato per l'esecuzione di processi all'interno dell'host stesso.

A differenza delle VM, un container non possiede al suo interno un intero Sistema Operativo, risultando più leggero di un ordine di grandezza (da GiB a centinaia di MiB) [5]. Esso condivide lo strato kernel dell'host. Questo è reso possibile grazie all'utilizzo di alcune funzionalità intrinseche di Linux, le quali consentono la creazione di aree isolate nel Sistema Operativo senza l'utilizzo dell'infrastruttura con hypervisor.

Analogamente alla virtualizzazione hardware, un container crea un ambiente completamente indipendente dagli altri task in esecuzione sul sistema host. Difatti, i processi al suo interno si ritengono gli unici detentori delle risorse hardware disponibili su un server.

L'avvio di container può essere effettuato utilizzando direttamente le funzioni di virtualizzazione fornite dal kernel Linux es. namespace (Sezione 2.2.1) o mediante opportuni software, denominati *Container Engine*. Questi offrono la possibilità di creare e gestire container senza la necessità di dover interagire direttamente con API del kernel.

La condivisione di un solo kernel tra più container rende l'isolamento fornito dalla virtualizzazione leggera più lasco. Difatti, l'accesso all'hardware da parte di un container può essere effettuato direttamente senza l'impiego di hypervisor. Questo rappresenta un vantaggio dal punto di vista prestazionale. Difatti, i container costituiscono una forma di virtualizzazione più performante rispetto alle VM [6].

Nella Figura 2.1 è possibile osservare gli strati che separano le applicazioni dal livello hardware in caso sia di VM che di container. Si noti come il Container Engine non rappresenti uno strato strettamente necessario per l'utilizzo di container.

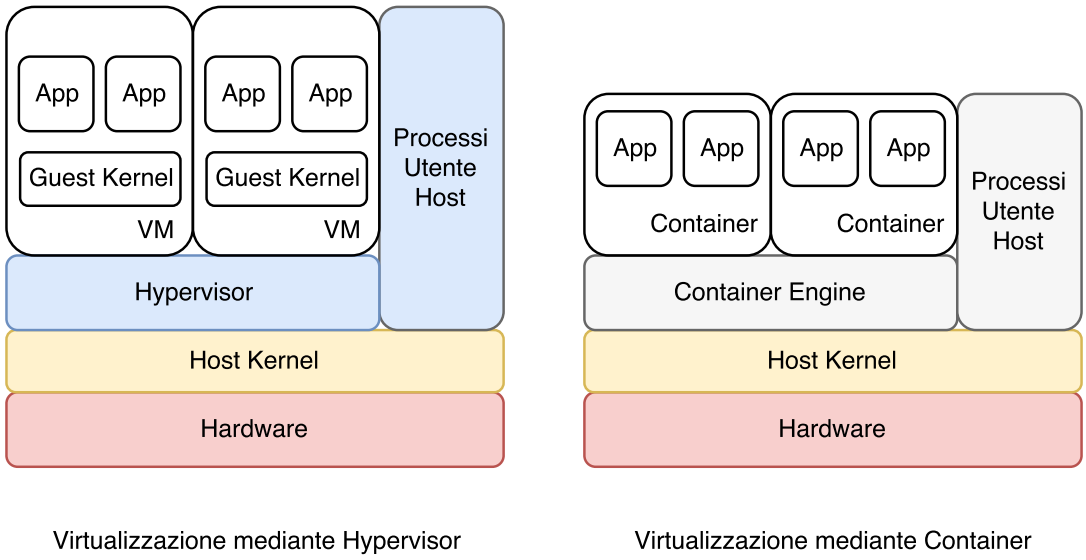

Figura 2.1. Struttura virtualizzazione hardware e virtualizzazione leggera.

2.2 Anatomia di un Container

Un container può essere generalmente definito come un ambiente isolato all'interno del quale sono eseguite una o più applicazioni [8]. Essi sono denotati come base per l'implementazione di un'architettura a microservizi [9].

All'interno del Sistema Operativo non esiste alcuna struttura dati che identifichi un container. Esso è un processo in esecuzione nello *user-space* [10]. Per quanto concerne la sua struttura

interna, l'architettura di esecuzione di un container sfrutta alcune caratteristiche del kernel Linux. Esse costituiscono i fondamenti della virtualizzazione leggera. In questa sezione sono esaminati *Namespace*, *Cgroup* e *Root Capability*.

2.2.1 Namespace

Il concetto di Namespace è stato introdotto per la prima volta in un Sistema Operativo nel Sistema Operativo Plan 9 [11] ed è stato implementato nel kernel Linux a partire dalla versione 2.4.19. Un Namespace permette di astrarre una risorsa del sistema in modo tale che essa sia visibile soltanto ai processi all'interno di esso [12]. Costituisce uno dei fondamenti della virtualizzazione leggera dal momento che offre ad un gruppo di processi la visibilità di uso esclusivo delle risorse dell'host.

Le impostazioni di ogni Namespace sono disponibili all'interno di un sistema Linux nella directory `/proc`. Tale percorso è identificato come pseudo-filesystem, ossia un filesystem che permette di interfacciarsi con le strutture dati del kernel con le stesse funzioni utilizzate per la gestione di Input e Output [13].

Attualmente sono disponibili sette tipi di Namespace [12]:

- *IPC* Namespace;
- *Network* Namespace;
- *PID* Namespace;
- *Cgroup* Namespace;
- *Mount* Namespace;
- *UTS* Namespace;
- *User* Namespace.

Gli IPC Namespace sono adoperati per l'isolamento di risorse nell'*Inter Process Communication* (IPC). Essa consiste di meccanismi per consentire la comunicazione e condivisione di dati tra processi [14]. Ogni Namespace di questo tipo possiede una propria coda di messaggi non visibile ad altri processi all'infuori di esso.

I Network Namespace permettono l'isolamento delle risorse associate al networking. Nello specifico, essi consentono ad un container di possedere una propria interfaccia virtuale e di poter collegare le proprie applicazioni ad un range di porte univoco per ogni Namespace. Ad esempio, è possibile avviare diversi container ognuno dei quali esegue un Web Server in ascolto sulla porta 80. Tale scenario prevede che l'host possieda delle regole di routing che instradino i pacchetti dall'interfaccia fisica al *device* virtuale associato al container [15, Parte 1].

I PID Namespace consentono il raggruppamento di processi in modo tale che lo stesso PID (Process ID) possa essere riutilizzato. Analogamente alle porte di rete, questi identificativi seguono una numerazione differente all'interno di ogni Namespace. Il primo processo inserito in un PID Namespace possiede PID 1 e ricopre il ruolo di processo `init`: esso adotta processi orfani all'interno del Namespace e, in caso di una sua chiusura, il kernel del sistema host procede alla terminazione di tutti i suoi processi figli (mediante segnale *SIGKILL*). Dunque, questa struttura permette facilmente la migrazione di container in diversi ambienti di produzione senza dover curare eventuali conflitti nella numerazione dei processi [16].

I Cgroup Namespace permettono ad ogni processo all'interno di un Namespace di avere una visione isolata dei Cgroup attivi (Sezione 2.2.2).

Gli *UNIX Time-sharing System* (UTS) Namespace consentono di isolare due identificatori del Sistema Operativo: l'*hostname* e il nome di dominio all'interno del *Network Information Service* (NIS). Essi consentono l'esecuzione di script che basano il loro funzionamento su specifici hostname o *NIS domain name* [15, Parte 1].

I Mount Namespace forniscono, ad uno o più processi in un container, una vista isolata del filesystem dell'host. Tale vista include punti di mount, device fisici e virtuali. Questo tipo di Namespace fornisce una protezione naturale nei confronti degli altri succitati. Difatti, impedisce a container la conoscenza dei path sotto `/proc/{PID}` che descrivono la configurazione degli altri Namespace [7, Capitolo 7].

Gli User Namespace realizzano l'isolamento di identificativi e attributi di sicurezza. Permettono ad un processo di avere l'identificativo dell'utente `uid` e l'identificativo del gruppo `gid` differenti, a seconda che lo si veda dall'interno o dall'esterno del Namespace. Ad esempio, un processo può avere `uid=0` (utente `root`) all'interno e `uid>0` all'esterno. In questo modo, è possibile che un processo abbia privilegi di amministratore all'interno di un container e risulti non privilegiato per operazioni al di fuori di esso [17]. Gli User Namespace isolano anche i *kernel keyring*, strutture dati utilizzate per la gestione delle chiavi di autenticazione e cifratura nel kernel. In assenza di questo tipo di Namespace le azioni svolte da un container su un keyring risulterebbero visibili anche ad altri container.

È opportuno notare che l'implementazione dei Namespace non ricopre alcune aree del kernel Linux, introducendo intrinsecamente una vulnerabilità di *Information Exposure*, ossia l'esposizione intenzionale o non intenzionale di informazioni ad attori non autorizzati all'accesso [18]. Alcune aree del già citato pseudo-filesystem `/proc` non sono isolate e permettono ad eventuali attaccanti di studiare le vulnerabilità di una macchina target.

2.2.2 Cgroup

I Cgroup (Control group) rappresentano un meccanismo per il raggruppamento di processi in una gerarchia, in cui differenti tipi di risorse possono essere limitati e monitorati [19]. Attualmente coesistono due versioni di Cgroup e questa sezione si limita ad esaminare la più recente.

L'architettura dei Cgroup v2 si basa essenzialmente su due elementi: il *core* e i *controller*. Il primo realizza una divisione gerarchica tra processi, mentre i secondi si occupano della distribuzione di una data risorsa attraverso la gerarchia creata dal primo. Ogni processo appartiene esclusivamente ad un Cgroup. I thread e i processi figli sono inseriti nello stesso del padre.

I controller possono essere abilitati o meno su di un Cgroup; qualora avvenga l'attivazione, tutti i processi appartenenti ne subiscono immediatamente l'effetto. Attualmente è disponibile il supporto per tre controller [20]:

Memoria. Traccia e limita l'utilizzo della memoria utente, le strutture dati del kernel e i buffer dei socket TCP.

PID. Limita il numero di PID disponibili ad un processo padre, vietando l'avvio di altri processi figli al raggiungimento di una soglia predefinita.

I/O. Regola la distribuzione di risorse di Input e Output in termini di larghezza di banda o numero di operazioni al secondo (IOPS).

Inoltre, sono disponibili diversi schemi per la distribuzione delle risorse. Ad esempio, è possibile assegnare un limite per una determinata risorsa a tutti i processi in un cgroup. La somma di tutti i limiti non deve oltrepassare la quantità assegnata al processo padre. Diversamente, si può convenire per una distribuzione pesata sull'importanza dei processi, in modo tale da favorire un task rispetto ad un altro [21].

L'utilizzo dei Cgroup permette significativi miglioramenti in termini di efficienza e sicurezza, dal momento che consente di allocare e gestire risorse in maniera granulare. Inoltre, la loro struttura ad albero richiama fortemente la gestione dei processi in Linux.

Sebbene possano sembrare simili, i Cgroup svolgono funzioni complementari a quelle dei Namespace: i primi si occupano di garantire una corretta ripartizione dell'hardware, mentre i secondi assicurano che i processi si considerino come unici detentori di una risorsa.

2.2.3 Root Capability

La violazione del principio di *least privilege* [22] si verifica nel momento in cui un processo è lanciato in esecuzione con privilegi non necessari all'adempimento delle sue funzioni. Ad esempio, si verifica nell'avvio di un programma con l'intero set di privilegi di amministratore, nonostante il numero di operazioni privilegiate richiesto sia estremamente ridotto. Esso aumenta notevolmente la probabilità che si verifichino interazioni malevoli tra un task e il suo ambiente di esecuzione.

Il concetto di Linux Root Capability è stato presentato a partire dalla versione 2.2 del kernel come uno strumento per partizionare i privilegi dell'utente `root` [7, Capitolo 5], superando la tradizionale suddivisione tra utente privilegiato (`uid=0`) e non privilegiato (`uid>0`).

Una Root Capability è un attributo specifico per ogni thread, implementato tramite un bit associato al file eseguibile salvato nel filesystem. Ogni bit indica un determinato privilegio. L'insieme delle Root Capability a disposizione di un generico task è raccolto in una struttura denominata *bitmap*. Utilizzando questo approccio, qualora un processo cerchi di effettuare una determinata operazione privilegiata, il Sistema Operativo può controllare se esso sia autorizzato ad effettuarla consultando la relativa bitmap. Dunque, il controllo risulta più granulare e non più basato sull'identificativo `uid`.

A titolo d'esempio, si consideri l'esecuzione di un Web Server. Il binding della porta 80 richiederebbe l'avvio dell'eseguibile come *super-utente*, dal momento che un sistema operativo basato su kernel Linux considera privilegiate le porte con identificativo minore di 1024. Tuttavia, al fine di non consentire al processo di avere pieni poteri sul sistema, è possibile settare la Root Capability `CAP_NET_BIND_SERVICE`; in questo modo, eventuali vulnerabilità nel codice sorgente espongono una minore superficie di attacco.

Sebbene le Root Capability permettano di seguire il succitato principio di minor privilegio, è indispensabile evidenziare che vi sono Root Capability più pericolose di altre. Ad esempio, l'assegnazione di `CAP_SYS_MODULE` ad un task consente l'avvio e l'arresto di moduli del kernel. Un utilizzo improprio di questo privilegio potrebbe condurre facilmente un malware di tipo *rootkit* ad un attacco di *Privilege Escalation*. Questo si concretizza quando un soggetto ottiene più risorse o funzionalità di quelle consentitegli, grazie al quale compie operazioni privilegiate a cui non è autorizzato [23].

Al contrario, `CAP_WAKE_ALARM` assicura esclusivamente la possibilità di impostare un avvio programmato del sistema. Pertanto, al fine di rendere l'approccio descritto in questa sezione utile ed efficiente, è necessario effettuare un'accurata selezione dei privilegi necessari ad ogni task.

Nella Tabella 2.1 è riportata una lista di Root Capability con le relative funzionalità, il cui abuso in un sistema di virtualizzazione leggera è ritenuto maggiormente rischioso [7, Capitolo 5]:

Root Capability	*Descrizione*
`CAP_SYS_MODULE`	Avvio e arresto di moduli kernel arbitrari
`CAP_SYS_ADMIN`	Avvio di altre capability. Controllo globale del sistema
`CAP_NET_ADMIN`	Modifica delle interfacce di rete dell'host, firewall e routing table
`CAP_SYS_ROOT`	Cambia la root directory virtuale di un processo
`CAP_SETUID`	Modifica dell'identificativo `uid` di un processo
`CAP_SETGID`	Modifica dell'identificativo `gid` di un processo
`CAP_RAW_IO`	Accesso a device fisici dell'host

Tabella 2.1. Principali Root Capability.

2.2.4 Principali tecnologie

Le implementazioni attuali di virtualizzazione leggera distinguono due diverse tipologie di container: *Application container* e *Machine (OS) container*. La prima si focalizza su container minimali, non persistenti e *single-purpose*. Il loro ciclo di esecuzione non prevede la modifica a runtime della

configurazione. La seconda prevede un utilizzo di container come VM *leggere*, in grado di evolversi e rimanere in esecuzione per lungo tempo [24]. Le principali soluzioni che implementano Machine container sono *LXC* [25] e *LXD* [26]; *Docker* [8] e *rkt* [27] sono i principali protagonisti nel settore degli Application container.

LXC

LXC è un'interfaccia utente facente parte del più ampio progetto *LinuxContainers* [28]. LXC permette di utilizzare le funzioni di isolamento del kernel fin qui descritte tramite un API (nella libreria `liblxc`). L'API può essere utilizzata mediante diversi linguaggi di programmazione e offre strumenti per la creazione e gestione dei container. Pertanto, l'impiego di LXC solleva l'utente dal compito di dover interagire direttamente con meccanismi a basso livello per la creazione di un ambiente di esecuzione isolato.

Con LXC si indica per estensione anche una tecnologia di OS container. Essa offre un ambiente di esecuzione simile a quello di una VM, senza l'overhead derivante dalla virtualizzazione hardware e l'utilizzo di un kernel separato [29]. LXC supporta il concetto di container non privilegiati, ossia container in esecuzione senza alcun privilegio di `root`. Richiede l'implementazione di User Namespace, come visto in Sezione 2.2.1.

LXD

Il progetto LXD, fondato e guidato da *Canonical Ltd*, si propone di offrire all'utente un ambiente simile a quello delle VM [24]. Alla base di esso si ritrova LXC, del quale LXD può essere inteso come un'estensione. Nello specifico, la struttura di LXD prevede l'utilizzo di un processo in background, detto *daemon*. Quest'ultimo espone un API REST che si interfaccia con `liblxc`. Nell'ambiente LXD è presente il progetto *Nova LXD* [30], il quale permette una piena integrazione con OpenStack [31], un framework utilizzato per la gestione distribuita di computazione, storage e networking all'interno di un datacenter.

rkt

L'engine rkt, rilasciato nel Dicembre 2014, prevede un'esecuzione di Application container orientata principalmente ad ambienti di produzione Cloud. Offre una facile integrazione con orchestratori quali Kubernetes [32]. Con quest'ultimo, inoltre, rkt condivide il concetto di *pod*. Un pod è un gruppo di uno o più container e rappresenta l'unità minima di esecuzione in rkt.

L'avvio di container avviene direttamente dalla *Command Line Interface* (CLI), senza l'intermediazione di un processo daemon. Infine, rkt include include il supporto per SELinux (Sezione 2.6.1) e il controllo dello stato e della configurazione dei container all'interno di un *Trusted Platform Module* (TPM).

Un TPM è un microchip che implementa funzioni di sicurezza, quali la generazione di chiavi, operazioni crittografiche (cifratura, hash crittografico) e memorizzazione *sicura* di dati all'interno di registri hardware, detti *Platform Configuration Registers* (PCR). Tra questi dati rientrano password, certificati e chiavi di cifratura. Il TPM consente la modifica dei PCR esclusivamente attraverso un'operazione crittografica detta *estensione*. I PCR possono essere utilizzati per memorizzare le misure della piattaforma in fase di avvio, così da consentire l'*attestazione* dello stato della piattaforma da parte di un *third-party verifier*. Inoltre, il TPM è in grado di garantire l'accesso ad un insieme di dati del sistema in funzione di uno stato specifico della piattaforma, mediante un processo detto *sealing* [33]. Le specifiche del TPM sono state pubblicate dal *Trusted Computing Group* (TCG), un consorzio di diverse aziende nel settore dell'industria informatica. La versione 2.0 della TPM Library Specification è stata approvata dall'*International Organization for Standardization* (ISO) e dall'*International Electrotechnical Commission* (IEC) ed è stata standardizzata come *ISO/IEC 11889:2015, Parts 1-4* nel 2015 [34].

Docker

La piattaforma Docker è trattata come caso di studio nell'ambito degli Application container nella Sezione 2.3.

2.3 Caso di studio: Docker

Docker è un software sviluppato inizialmente dalla dotCloud,Inc., ora Docker,Inc. [35]. È stato rilasciato come progetto open-source nel 2013 e rappresenta attualmente la principale tecnologia nell'ambito della virtualizzazione leggera [36]. Attualmente alla versione 17.06, è disponibile sia come software gratuito (CE), sia come prodotto commerciale per le aziende (EE).

L'ecosistema Docker è costituito da diverse soluzioni software, le quali contemplano sia la gestione che l'orchestrazione di container.

Il core della piattaforma Docker è il *Docker Engine*, un'applicazione client-server che prevede i seguenti componenti [37]:

- il *Docker Daemon*;
- il *Docker Client*;
- una REST API (Docker API).

Il Docker Daemon è un processo in ascolto sulla Docker API e si occupa della gestione di immagini, container, volumi e reti. Il Docker Client traduce gli input di una CLI in richieste REST e si interfaccia con il Docker Daemon.

Nelle sue prime versioni, Docker utilizzava LXC per interfacciarsi con le funzioni di isolamento del kernel esposte nella Sezione 2.2. Tuttavia, a partire dalla versione 0.9, LXC è stato sostituito dalla libreria `libcontainer`, nata dal desiderio della community Docker di avere il proprio formato di virtualizzazione indipendente da terze parti [38].

2.3.1 Componenti principali

La Figura 2.2 illustra l'interazione del Docker Engine con diversi componenti, di seguito analizzati.

Immagine

Il punto di partenza per l'avvio di un container è l'*Immagine*, un template in sola lettura costituito da una struttura stratificata, come illustrato dalla Figura 2.3. Ogni immagine si presenta come un insieme di più livelli (in sola lettura), ognuno dei quali la personalizza ulteriormente.

Le direttive per la realizzazione di un'immagine sono espresse mediante una sintassi specifica all'interno di un *Dockerfile*. Nello specifico, ogni comando all'interno di un Dockerfile costituisce uno strato nell'immagine finale. Il processo di costruzione di un'immagine a partire da un Dockerfile prende il nome di *build*.

La stratificazione è permessa dal filesystem *AUFS*, una particolare tecnologia che permette di sovrapporre diverse directory e unificarle sotto un singolo punto di mount [8].

Infine, la natura a livelli di un'immagine offre la possibilità di creare nuove immagini a partire da immagini preesistenti, semplicemente sovrapponendo a queste ultime ulteriori livelli.

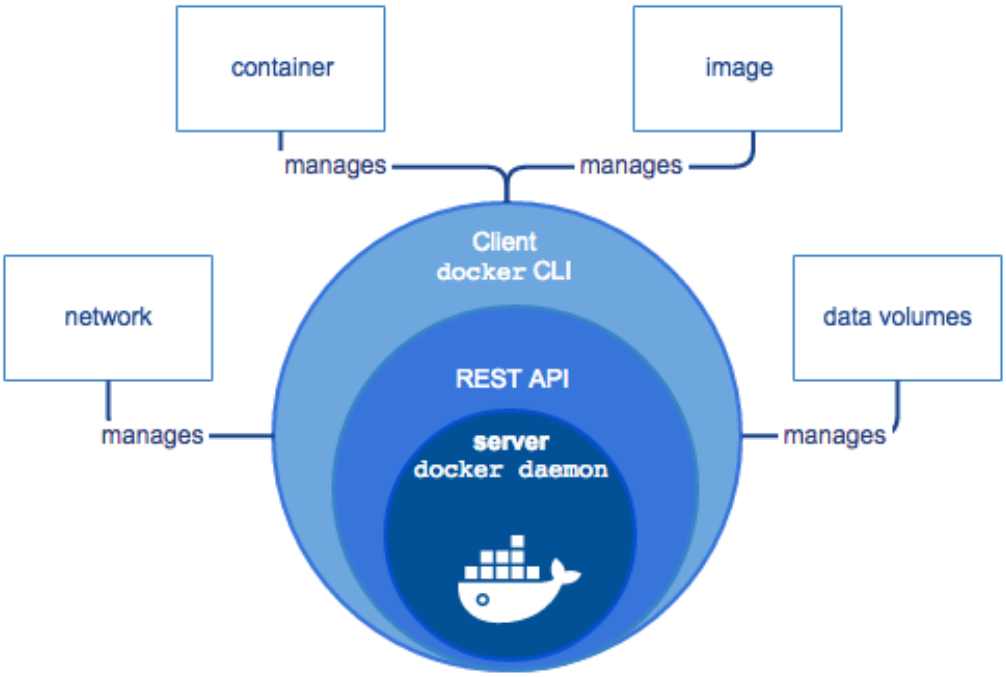

Figura 2.2. Principali componenti dell'architettura Docker (fonte: docker).

Container

In Docker, un *container* è definito come un'istanza in esecuzione di un'immagine. Esso può essere controllato interamente mediante la CLI di Docker Client o l'interazione diretta con la Docker API.

Dal momento che le immagini da cui derivano sono in sola lettura, i container utilizzano una strategia di *Copy on Write* (CoW) per effettuare modifiche a *runtime* dei dati di partenza.

La CoW è un meccanismo di ottimizzazione che evita di replicare il contenuto di un'entità condivisa. Nello specifico, essa prevede che un oggetto utilizzato da più processi non sia copiato nello spazio di memoria di ogni processo. Ognuno di essi condivide una copia dell'entità iniziale fin quando non necessita di modificarla. In tal caso, provvede prima a copiarla nel proprio spazio di memoria, al fine di non propagare la sua modifica agli altri utilizzatori.

Nel caso di Docker la CoW si realizza mediante l'aggiunta di un ulteriore livello ad un'Immagine, questa volta in lettura e scrittura. Tale livello, a meno di utilizzare un meccanismo di storage apposito es. Volume, è rimosso all'arresto del container.

La Figura 2.3 illustra la struttura del filesystem AUFS in Docker, evidenziandone gli strati in sola lettura e lo strato di CoW, denominato *Container layer*. Il punto di *Union mount* consente di vedere le diverse directory presenti negli strati in sola lettura e nel Container Layer sotto un singolo punto di mount, ossia come se appartenessero tutte ad una sola directory.

Volume

Un *Volume* è una directory all'interno di uno o più container che oltrepassa il filesystem AUFS [8]. Esso consente di ottenere la persistenza dei dati oltre il ciclo di vita di un container senza apportare alcuna modifica all'immagine di partenza.

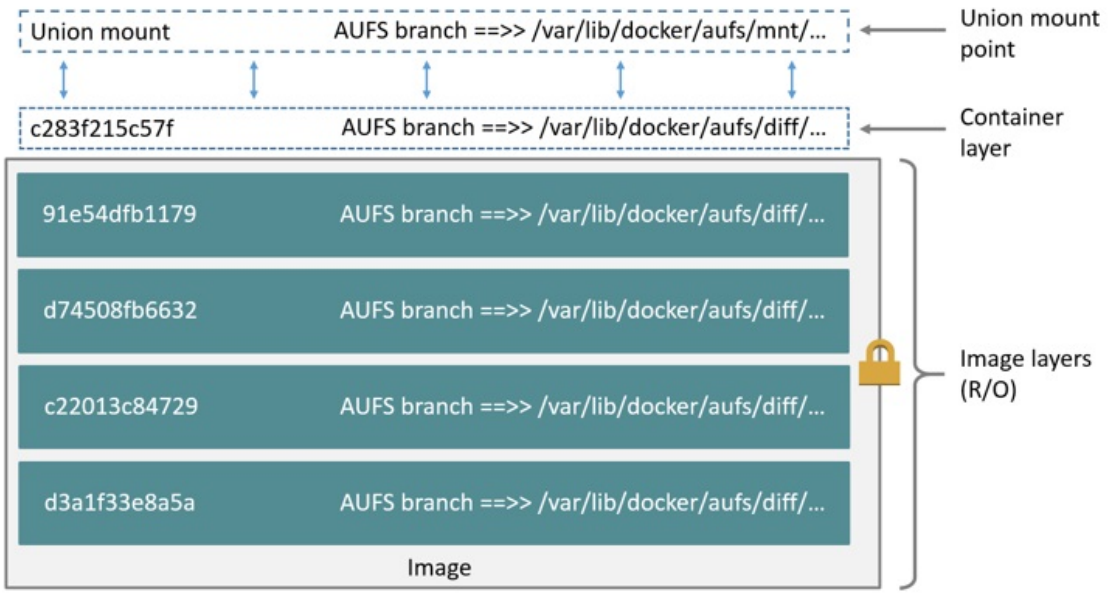

Figura 2.3. Struttura filesystem AUFS in Docker (fonte: docker).

I volumi costituiscono anche una modalità di condivisione di dati tra diversi container o tra un container e l'host. La mappatura di un Volume nel filesystem dell'host può avvenire sotto una directory di default o su una qualsiasi specificata appositamente dall'utente. Dal momento che tale meccanismo rappresenta un'interazione tra entità differenti, è opportuno tener conto delle potenziali vulnerabilità espresse nella Sezione 2.5.4.

2.3.2 Docker Registry

Un *Docker Registry* è un sistema per il salvataggio e la distribuzione di immagini. Docker offre la possibilità di interagire con immagini presenti nel *Docker Hub*, un Registry che permette di creare repository pubblici e privati. In alternativa, è lasciata all'utente la possibilità di creare un Registry personale per avere un controllo diretto sulle immagini salvate nel sistema, controllarne la distribuzione e integrarle in un flusso di sviluppo [8].

Docker fornisce la possibilità di utilizzare il protocollo TLS [39] e *basic authentication* per l'accesso ai Registry. Nella Sezione 2.7 sarà considerato il problema dell'affidabilità e della distribuzione delle immagini.

2.4 Il problema della sicurezza

Nella Sezione 2.1 sono state introdotte le principali differenze tra virtualizzazione hardware e leggera. È emerso come un isolamento più limitato, realizzato mediante la condivisione del kernel del Sistema Operativo, possa abbattere il degrado delle prestazioni ed il consumo delle risorse riscontrato nella virtualizzazione hardware. Tuttavia, è fondamentale sottolineare che l'accesso al kernel da parte di container introduce diverse sfide nel campo della sicurezza, finora non considerate.

Un primo aspetto da esaminare è l'interazione non prevista dell'entità guest con l'host. Tale fenomeno, denominato *escape*, consiste nella violazione dei principi di isolamento e può provocare la compromissione del sistema sottostante [7, Capitolo 1]. L'impiego di VM comporta che un attaccante, dopo aver acquisito il controllo di un'istanza, possa propagare l'attacco all'host solo dopo aver oltrepassato il kernel della macchina virtuale e l'hypervisor specifico della tecnologia in adozione. Al contrario, la compromissione di un container offre ad un utente malevolo la possibilità

di interfacciarsi direttamente con il Sistema Operativo dell'host e le sue risorse. Dunque, è evidente come la mancanza degli strati software descritti nel primo caso possa semplificare un attacco rivolto all'host.

Una seconda considerazione riguarda l'utilizzo delle risorse hardware. Il setup di una VM prevede l'allocazione di un set predefinito di risorse fisiche che permettono l'esecuzione dell'istanza. Tale assegnazione, nella maggior parte dei casi non modificabile a *runtime*, assicura che la saturazione delle risorse di una VM non provochi lo *starvation* di altre. Tale fenomeno consiste nell'impossibilità di un processo di ottenere l'hardware sufficiente per procedere all'esecuzione. Dal lato opposto, essendo i container a diretto contatto con la macchina host, l'isolamento hardware non è garantito. Ad esempio, un ciclo infinito all'intero di un container potrebbe provocare l'esaurimento della RAM dell'host e, nel caso peggiore, un *kernel panic*, ossia un crash, nel sistema host.

Infine, la virtualizzazione leggera enfatizza il problema della distribuzione delle immagini. A differenze di altri software, VM e container sono facilmente portabili. Essi possono essere *migrati* da un ambiente ad un altro senza dover procedere ad una nuova installazione nel sistema. Questo è possibile grazie al concetto di immagine. Un'immagine (si veda la Sezione 2.3.1) è il punto di partenza per l'esecuzione di VM/container; dunque, risulta evidente come la sua affidabilità sia indispensabile per il suo utilizzo, soprattutto nel Cloud Computing. Nel caso di container, tale problema risulta accentuato poiché le ridotte dimensioni velocizzano la distribuzione di immagini malevoli.

Nella maggior parte dei casi, la dimensione della superficie d'attacco dell'ambiente di produzione può essere notevolmente limitata con il solo rispetto di *best-practice* nella configurazione. Ovviamente, la sicurezza del software rimane indispensabile per arginare le attività di un attaccante.

Nei paragrafi seguenti seguirà una digressione su best-practice e strumenti per la sicurezza della virtualizzazione leggera, applicabili a livello utente e kernel. Infine, si discuterà del problema della distribuzione delle immagini di container.

2.5 Sicurezza di Docker nello spazio utente

Nella Sezione 2.4 è stato anticipato che la superficie di attacco in un ambiente di produzione basato su container può essere notevolmente ridotta mediante una corretta configurazione dell'ambiente di esecuzione.

La presenza di impostazioni di default non adeguate costituisce una delle principali cause di attacco a scenari di virtualizzazione leggera [7, Capitolo 7]. Tra queste, si ritrova in primo luogo l'esecuzione di processi all'interno dei container con privilegi di amministratore. Difatti, se non diversamente specificato, un'applicazione virtualizzata in ambiente Docker mantiene lo `uid` ereditato dal Docker Daemon, ossia 0.

Il *Center for Internet Security* (CIS) [40] ha rilasciato diversi benchmark su Docker [41], al fine di segnalare best-practice per la configurazione dell'ambiente di virtualizzazione. Nel corso della trattazione si farà riferimento al *Docker 1.13.0 Benchmark* [42].

Nella presente Sezione si considerano le principali raccomandazioni per l'utilizzo dei container Docker nello spazio utente, rimandando alla Sezione 2.6 le configurazioni inerenti il sistema operativo.

2.5.1 Configurazione del Docker Daemon

Il Docker Daemon, necessario per l'avvio di container e applicazioni con Docker, richiede privilegi di amministratore per la sua esecuzione. Tali privilegi possono essere acquisiti da un utente che rientra nei *sudoer* del Sistema Operativo o nel gruppo utenti `docker`. Nei Sistemi Operativi Linux, un sudoer è un utente autorizzato ad effettuare operazioni privilegiate sul sistema mediante il comando `sudo`. Dunque, è necessario che l'accesso al Docker Daemon sia riservato esclusivamente ad utenti fidati [8].

A titolo d'esempio, si consideri il seguente scenario. L'utente malevolo *non privilegiato* `mallory` appartiene al gruppo utenti `docker` e può interagire con il Docker Daemon di un sistema di produzione. Egli cerca di accedere al file `/etc/shadow` dell'host. Tale file contiene diverse informazioni sugli account presenti nel sistema, tra cui le password in forma cifrata. Per questo motivo, si consiglia di mantenerlo inaccessibile ad utenti non amministratori [43]. Non essendo un utente privilegiato, il tentativo di lettura di `mallory` è bloccato dal *Discretionary Access Control* (DAC) di Linux mediante un messaggio di errore. Tuttavia, essendo parte del gruppo `docker`, `mallory` può provvedere ad avviare un container che, mediante un volume, mappi in modalità lettura/scrittura la directory radice `/` dell'host. Difatti, il Docker Daemon, essendo in esecuzione con privilegi di `root`, consente operazioni di mount senza alcuna limitazione sia in termini di directory che di permessi. Si realizza così un attacco di *privilege escalation*. Il container appena creato può accedere con `uid=0` ai file del filesystem dell'host desiderati da `mallory`. Inoltre, avendo qualsiasi tipo di accesso ai file del Sistema Operativo, `mallory` può altresì cercare di eliminare le tracce della sua azione non autorizzata.

Per mitigare questo tipo di attacco è necessario l'utilizzo di un meccanismo di controllo accessi *Mandatory Access System* (MAC), descritto nella Sezione 2.6.1.

Altri esempi di interazione da parte di un utente malevolo con il Docker Daemon possono ricondurre a diverse altre categorie di attacchi, quali *Denial of Service* (DoS) e *Defacement*.

Un attacco DoS ha come obiettivo quello di rendere un servizio (o una risorsa) non disponibile all'utilizzo per cui è stato disposto. A differenza del *Distributed Denial of Service* (DDoS) prevede che vi sia un unico vettore di attacco. Il DoS sfrutta la presenza di vulnerabilità all'interno di codice sorgente, configurazioni di rete o risorse. Lo sfruttamento di tali vulnerabilità avviene mediante un flusso enorme di richieste non legittime che mirano alla saturazione di una determinata risorsa es. RAM. Ad esempio, un processo che satura la RAM disponibile in un host con lo scopo di rendere inutilizzabile la macchina effettua un attacco DoS, poiché impedisce in questo modo l'utilizzo dell'hardware da parte di processi legittimi.

Un attacco di Defacement mira al cambiamento non autorizzato della veste grafica di una pagina Web o di un sito. Si concretizza mediante l'iniezione di contenuti quali pubblicità o immagini non appropriate o la sostituzione dell'intero sito con contenuti non appropriati [44].

Docker Engine consente l'interazione con il Docker Daemon mediante socket TCP o UNIX. Questi ultimi rappresentano una famiglia di socket utilizzabili per la comunicazione tra processi all'interno dello stesso Sistema Operativo Linux [45]. Qualora si utilizzasse un socket TCP, il CIS raccomanda un accesso autenticato mediante protocollo TLS. Difatti, in assenza di autenticazione, la sola possibilità di essere contattato dall'esterno costituirebbe una minaccia per l'host stesso.

2.5.2 Configurazione di rete

Docker Engine prevede nelle impostazioni di default che i container siano avviati in possesso della Root Capability `CAP_NET_RAW` [7, Capitolo 5]. Tale privilegio consente l'accesso a socket sia di tipo *PACKET* sia *RAW*, ossia socket che inoltrino pacchetti dotati o meno della formattazione data da un livello di trasporto es. TCP.

L'esecuzione di container appartenenti a diversi clienti sullo stesso host è una soluzione comune alle aziende che offrono servizi di tipo *Platform as a Service* (PaaS) [46]. Tali aziende hanno come core business la fornitura di piattaforme di sviluppo ed esecuzione di applicazioni.

Di default, in Docker, il networking tra container prevede che tutte le istanze condividano lo stesso Virtual Bridge `docker0`, come illustrato in Figura 2.4. Un Virtual Bridge è un software che svolge all'interno di un Sistema Operativo le stesse funzioni svolte da un Bridge fisico. Le implementazioni principali sono Linux Bridge [47] e Open vSwitch [48].

La condivisione di `docker0` e la presenza di `CAP_NET_RAW` rende container presenti all'interno dello stesso host vulnerabili ad un attacco di tipo *Man In The Middle* (MITM).

Il MITM consiste nell'interposizione di un attaccante nella comunicazione tra due attori. Una volta inseritosi, egli può provvedere alla lettura, inserimento, modifica o cancellazione dei dati in

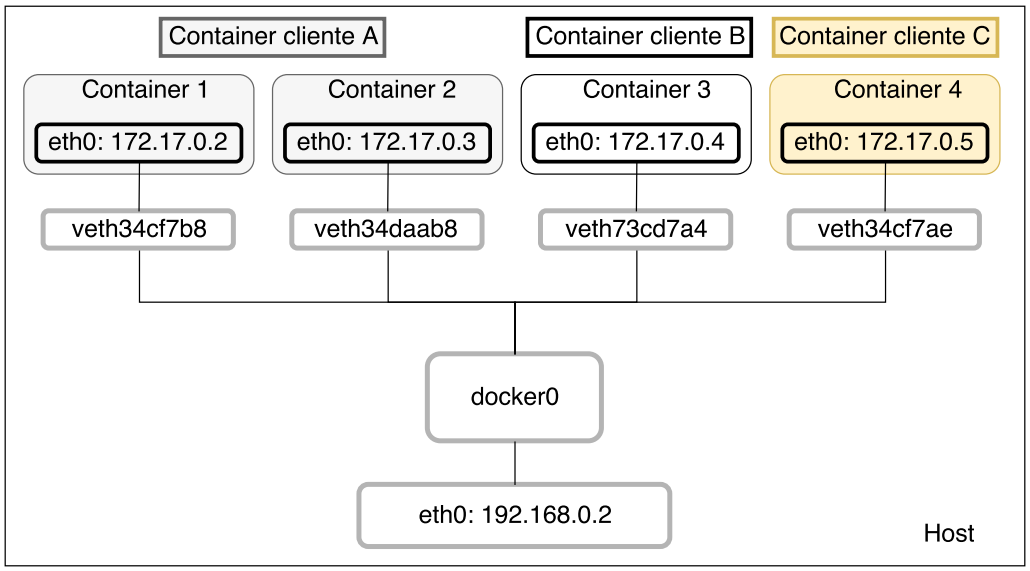

Figura 2.4. Default networking in Docker in ambiente PaaS.

transito. Il tipo di attacco MITM in esame prende il nome di *ARP Spoofing* o *ARP Poisoning*. Nello stack di rete TCP/IP, il protocollo ARP [49] è utilizzato per la risoluzione del MAC address di un nodo. Nel caso di virtualizzazione leggera, è impiegato dai Virtual Bridge per consegnare frame Ethernet al container corrispondente. L'attacco di ARP Poisoning prevede che un attaccante *avveleni* la cache ARP delle schede di rete di due vittime. Nello specifico, indica ad entrambe il suo indirizzo MAC come indirizzo della controparte. In questo modo, le schede di rete delle vittime invieranno frame inconsapevolmente all'attaccante, il quale dopo aver agito sui contenuti, si occupa di recapitarli al legittimo destinatario.

Trattandosi di un protocollo di livello due nello stack TCP/IP, la mitigazione di questa vulnerabilità non può essere fatta tramite regole a livello tre, ossia routing IP. Pertanto si può procedere in due modalità differenti.

Un primo approccio consiste nella disabilitazione della capability `CAP_NET_RAW`, al fine di impedire al container l'utilizzo di socket RAW. Tuttavia, la mancanza di questi non consente ai container l'utilizzo di comandi come `ping`, i quali potrebbero risultare necessari ai fini di debug.

Una seconda soluzione consiste nella creazione di un Virtual Bridge per ogni container o ogni gruppo di container ritenuti fidati. In questo caso, mediante *packet filter* è possibile isolare domini di gruppi differenti [46].

Networking in Host Mode

Tra le possibili impostazioni di networking, i container possono essere utilizzati nel cosiddetto *host mode*. Esso prevede la condivisione del Network Namespace dell'host [8]. Dal punto di vista prestazionale, questa modalità si dimostra più performante. Difatti, i container possono condividere lo stack di rete dell'host senza attraversare livelli di virtualizzazione [50].

L'utilizzo dello stesso stack di rete, tuttavia, consente ai container l'accesso a diverse risorse, tra cui servizi di sistema in ascolto su UNIX Socket. Inoltre, consentirebbe l'apertura di porte con identificativo minore di 1024, ritenute privilegiate nei sistemi Linux per motivi di sicurezza. Pertanto, il CIS raccomanda di evitare l'utilizzo di questa modalità.

2.5.3 Limitazione delle risorse

A differenza di quanto accade nelle VM, le applicazioni all'interno dei container possono interagire direttamente con il Sistema Operativo dell'host. Pertanto, se non opportunamente confinata, la

```
#include <unistd.h>

void main() {
  while(1) {
    fork();
  }
}
```

Figura 2.5. Fork Bomb in linguaggio C.

loro esecuzione può condurre facilmente alla saturazione di risorse.

PID

Si consideri l'identificativo PID. Il kernel Linux, su un sistema a 64 bit, prevede tramite la direttiva `PID_MAX_LIMIT` un numero massimo di identificativi pari a 2^{22}, oltre il quale non è più possibile avviare nuovi processi.

La piattaforma Docker prevede, nella configurazione di default, che i processi in esecuzione in un container abbiano un proprio PID Namespace, il quale nega loro la conoscenza degli altri processi in esecuzione nel sistema. Tuttavia, non si contempla l'utilizzo di Cgroup per il contenimento del numero di processi. In questo scenario, un container malevolo potrebbe provocare un attacco DoS all'host mediante un *fork bomb*. Un fork bomb è un malware che mira alla saturazione delle risorse di un sistema replicandosi.

Il codice sorgente riportato in Figura 2.5 dimostra la creazione di un fork bomb in linguaggio C. Esso consiste in un ciclo infinito che esegue una `fork()`. Tale funzione prevede la creazione di un processo figlio a partire da un esatto duplicato del padre. Dal momento che il codice è ricorsivo, ogni processo figlio ripete la stessa operazione, decretando un aumento esponenziale dei processi in esecuzione.

La saturazione del numero di PID disponibili causa inevitabilmente un crash del sistema host, dal momento che non è possibile in nessun caso procedere alla terminazione dei processi creati (il comando `kill` è esso stesso un processo). Il CIS consiglia di configurare il PID Cgroup per ogni container mediante l'utilizzo della direttiva `--pid-limit` all'avvio di ogni container, al fine di contenere eventuali fork bomb alla singola istanza compromessa.

Memoria

Le impostazioni di default in Docker [8] non prevedono un limite nel consumo di memoria RAM. La mancanza di una soglia massima nell'allocazione di RAM per container comporta una vulnerabilità sfruttabile mediante un attacco di tipo DoS.

Un'Immagine malevola potrebbe includere al suo interno l'esecuzione di script estremamente onerosi da un punto di vista della RAM. Consideriamo ad esempio lo script 2.6.

Il loop principale concatena ricorsivamente l'output del generatore random di numeri del kernel Linux. Dal momento che non vi è imposto alcun limite, in poco tempo il container provocherà nell'host un'eccezione di *Out of Memory* (OOM). La situazione di OOM, a seconda della configurazione del kernel, può portare alla terminazione forzata di altri container o ad una situazione di *panic*.

Il CIS consiglia in Docker l'utilizzo del flag `-m` per definire la massima quantità di memoria a disposizione di un container. In caso di violazione dei limiti, il kernel è autorizzato a procedere alla terminazione immediata di tale istanza. Tuttavia, la scelta di una soglia idonea può rivelarsi difficile con applicazioni onerose dal punto vista della memoria RAM, ad esempio *Java Virtual Machine* (JVM) [51]

```
#/bin/bash
for (( ; ; ));
do
        test="${test}$(cat /dev/random)"
done
```

Figura 2.6. Script Bash per consumo progressivo di memoria.

2.5.4 Filesystem

Analogamente alle VM, un container può condividere file con l'host, interagendo con il filesystem al di fuori del Mount Namespace (Sezione 2.3.1). Questo si dimostra utile nel momento in cui si desidera la persistenza di alcuni dati oltre il ciclo di vita di un container (file di log). Tuttavia, la scelta della directory del filesystem dell'host da montare all'interno del container è di fondamentale importanza. Difatti, se non specificato diversamente, il container avrà in tale directory permessi di lettura e scrittura.

Per la persistenza dei dati in Docker, il CIS suggerisce la creazione di volumi senza una mappatura diretta tra i filesystem dell'host e del container. In questo modo, è creata una directory all'interno del path `/var/lib/Docker/volumes` dell'host che preclude al container la conoscenza di altri file presenti nel sistema.

Inoltre, il CIS raccomanda l'utilizzo di un opportuno flag all'avvio di ogni container (`--read-only`) per escludere la possibilità di modifiche da parte di questo a file o cartelle al di fuori del volume definito.

2.5.5 User Namespace

All'inizio di questa Sezione è stato anticipato come le applicazioni all'interno di un container siano in esecuzione con privilegi di `root`. Questo può creare gravi problemi in casi di escape (Sezione 2.4).

Per arginare gli effetti derivanti da questa eventualità, la documentazione ufficiale di Docker [8] consiglia di cambiare l'utente che esegue i processi all'interno di un container da `root` a non privilegiato. Questo risultato si può ottenere in diversi modi, tra cui l'utilizzo della direttiva `USER` all'interno del Dockerfile.

Tale soluzione tuttavia non può essere considerata efficace in caso di immagini di cui non possiamo stabilire l'attendibilità (Sezione 2.7).

Pertanto, il CIS consiglia l'utilizzo degli User Namespace introdotti nella Sezione 2.2. Essi consentono di effettuare un mapping tra gli `uid` all'interno del container e gli `uid` all'esterno di esso. In questo caso, eventuali fenomeni di escape porteranno i processi avviati nel container ad agire come utenti non privilegiati all'interno dell'host.

In Docker, l'utilizzo di User Namespace non è previsto di default. È necessario che tale meccanismo sia attivato dall'utente, accedendo alle impostazioni del Docker Daemon e aggiungendo il flag `--userns-remap="`*`user`*`"`.

2.6 Sicurezza di Docker nel kernel

Nella presente Sezione si considerano le principali raccomandazioni per l'utilizzo dei container Docker nello spazio kernel.

2.6.1 SELinux

Security-Enhanced Linux (SELinux) è un modulo di sicurezza presente nel kernel Linux a partire dalla versione 2.6 che implementa un *Mandatory Access Control* (MAC). Il progetto SELinux è sostenuto dalla National Security Agency (NSA) statunitense e dalla SELinux community [52].

Al fine di comprendere le funzionalità e l'impiego di SELinux si introducono i concetti di *Discrectionary Access Control* (DAC) e MAC. Entrambi costituiscono modelli per il controllo degli accessi.

Il DAC utilizza il concetto di *Discrectionary Policy*. Tale tipologia di policy gestisce gli accessi sulla base dell'identità degli utenti e delle relative autorizzazioni. Queste ultime specificano, per ogni utente, i permessi di accesso che egli detiene su un particolare oggetto. Qualsiasi tipo di accesso, sia esso di scrittura, lettura o esecuzione, richiede il possesso della relativa autorizzazione [53].

Un tipico esempio di DAC è il meccanismo di permessi in un tradizionale filesystem UNIX. Esso prevede l'assegnazione di una *Access Control List* (ACL) ad ogni file, la quale indica quali privilegi sono consentiti su di esso ad ogni utente. L'ACL può essere modificata *a discrezione* del proprietario del file.

La centralità del concetto di identità, tuttavia, rende il DAC facilmente aggirabile. In molti sistemi, ogni processo in esecuzione per conto di un utente utilizza l'identità di quest'ultimo per accedere a qualsiasi risorsa. Pertanto, eventuali vulnerabilità del software possono essere sfruttate per impersonare l'utente all'interno del sistema [54].

Il MAC impiega il concetto di *Mandatory Policy*. Questa prevede la gestione degli accessi mediante classificazione di ogni entità presente nel sistema, sia essa un utente, processo o oggetto [53]. Tale classificazione avviene mediante l'utilizzo di opportune *label*. A differenza del DAC, il MAC non considera l'identità dichiarata da un processo, ma procede alle sue valutazioni solo sulla base delle label a cui è associato. Ad esempio, un processo P con label L1 può scrivere su un file F di tipo F1 soltanto se la policy consente a processi etichettati con L1 di scrivere su file con label F1. In caso contrario, l'accesso è negato.

Una Mandatory Policy può essere definita esclusivamente dall'amministratore di sistema. Pertanto, questo modello di accesso permette di colmare le vulnerabilità non contemplate dalla discrezionalità del DAC [55].

È opportuno precisare che i due modelli di accesso descritti non sono mutuamente esclusivi. Nello specifico, il MAC è effettuato solo in seguito ad un controllo DAC con esito positivo.

Nel caso di virtualizzazione leggera, SELinux può essere impiegato secondo il modello *Type Enforcement* (TE), per garantire la protezione dell'host e *Multi Category Support* (MCS) per l'isolamento tra container.

Type Enforcement

Il modello TE prevede di etichettare ogni processo con una label detta *dominio* e ogni oggetto con un'altra detta *tipo*. Esso valuta senza distinzioni tutti i processi nello stesso dominio e tutti gli oggetti dello stesso tipo [56].

Nel contesto di un container, Docker etichetta tutti i processi con il dominio `svirt_lxc_net_t` e tutti i file con il tipo `svirt_sandbox_file_t`. Dunque, i processi in esecuzione con `svirt_lxc_net_t` possono accedere in lettura/scrittura solo ai file etichettati con `svirt_sandbox_file_t`. In questo modo, SELinux consente dunque di prevenire l'attacco di privilege escalation descritto nella Sezione 2.5.1.

In aggiunta, previene possibili Information Exposure, dal momento che applica restrizioni sul filesystem dell'host anche in lettura (Sezione 2.5.4).

Multi Category Support

Il TE con SELinux vieta ad ogni container la possibilità di accedere ad oggetti non etichettati con il tipo `svirt_sandbox_file_t`. Tuttavia, esso non contempla l'eventualità di attacchi tra container.

Si consideri il seguente scenario con i container `docker1` e `docker2`. In TE, Docker assegna ai processi in `docker1` la stessa etichetta delle applicazioni in `docker2`. Dal momento che i file di entrambi i container rientrano nello stesso tipo, l'accesso di `docker1` ai contenuti di `docker2` risulta erroneamente lecito. Per scongiurare tali scenari, SELinux utilizza la modalità Multi Category Support.

MCS introduce un ulteriore strato di controllo ed è invocato nel caso sia il DAC che il MAC con TE abbiano consentito l'accesso del processo alla risorsa. Esso consiste nell'etichettare una risorsa con un label di *livello*.

Un label di livello ha la seguente struttura: `s0:c1,c3,c7`. La parte antecedente `:` rappresenta la *sensibilità* della risorsa, mentre la parte successiva indica la lista di *categorie* alle quali la risorsa appartiene [57]. I valori di sensibilità sono ordinati in maniera crescente, a partire dal meno sensibile `s0` al più sensibile `s15`. Gli identificativi delle categorie, invece, non sono ordinati e rappresentano l'appartenenza o meno della risorsa ad una determinata categoria.

Mediante l'indicazione del livello, SELinux può gestire gli accessi agli oggetti nei container pur essendo questi ultimi tutti dello stesso tipo `svirt_sandbox_file_t`. A titolo d'esempio si considerino tre container:

- `docker1` con etichetta di livello `s0:c1,c5`;
- `docker2` con etichetta di livello `s0:c5`;
- `docker3` con etichetta di livello `s0:c2,c5`.

Tutti i container hanno sensibilità pari a s0; pertanto, essa non costituisce un fattore discriminante per MCS. Il container `docker1` ha accesso solo alle risorse appartenenti *contemporaneamente* alla categorie `c1` e `c5`; il container `docker2` solo a quelle di categoria `c5` e il container `docker3` quelle di categorie `c2` e `c5`. Dal momento che i file all'interno di un container ereditano la stessa etichetta di livello del container, si evince che:

- `docker1` potrà accedere ai dati di `docker1` e `docker2`
- `docker2` potrà accedere ai dati di `docker2`;
- `docker3` potrà accedere ai dati di `docker2` e `docker3`.

Pertanto, l'applicazione di etichette di livello permette l'isolamento di risorse e processi appartenenti a container differenti, contribuendo alla prevenzione di attacchi *Cross-container*.

2.6.2 SECCOMP

Una *system call* è una particolare funzione richiamata da un programma in esecuzione nello user-space per richiedere un determinato servizio dal kernel del Sistema Operativo. Il Kernel Linux offre centinaia di system call. Tuttavia la maggior parte di esse non è adoperata nel ciclo di vita dei processi [58].

SECure COMputing (SECCOMP) è una struttura del kernel Linux che offre la possibilità di filtrare le chiamate di sistema, riducendo la superficie del kernel esposta ad un'applicazione. Dal momento che la condivisione del kernel espone a tecnologie di virtualizzazione leggera una vasta superficie d'attacco, l'utilizzo di SECCOMP può limitare la lista di syscall a disposizione di un container. Attualmente, Linux supporta due modalità per impiegare SECCOMP: *STRICT* e *FILTER*.

La modalità STRICT consente ad un thread chiamante solo quattro system call: `read()`, `write()`, `_exit()` e `sigreturn()`. Qualora un thread tentasse di effettuare altre system call riceverebbe immediatamente un signale di terminazione `SIGKILL` dal kernel. Tale modalità è consigliata per l'esecuzione di bytecode non fidato, ottenuto ad esempio tramite un socket [59]. Difatti, trova applicazione in piccoli processi come i motori di rendering in Browser Web. Al contrario, esso si dimostra difficilmente idoneo per l'esecuzione di container [7, Capitolo 8].

La modalità FILTER prevede l'utilizzo di un *Berkeley Packet Filter* (BPF). Quest'ultimo consiste in una struttura kernel per la cattura di pacchetti di rete ad alte prestazioni [60]. Nel caso di SECCOMP, il BPF non è utilizzato per esaminare i campi dei pacchetti IP, bensì per analizzare l'identificativo della system call richiamata da un processo e i suoi argomenti. L'abilitazione di questa modalità all'interno di un processo prevede di installare nel kernel un puntatore ad un BPF, il quale contiene la lista di system call concesse o bloccate, a seconda che si usi un approccio di tipo whitelist o blacklist. Il BPF è invocato ad ogni system call del processo su cui è attivato. A differenza del caso precedente, un'invocazione ad una chiamata di sistema non concessa può essere gestita in diversi modi: a seconda della situazione si può preferire di limitarsi al solo log o provvedere alla terminazione immediata del processo chiamante. Ai fini di sicurezza, la lista di system call concesse ad un processo non può essere allungata a *runtime*, ovvero un filtro impostato all'avvio del processo può subire modifiche successivamente solo per essere più selettivo e non più permissivo.

Sebbene la modalità FILTER possa sembrare più idonea ad un ambiente di virtualizzazione leggera, è opportuno considerare che la scelta del numero minimo di syscall costituisce un problema oneroso dal punto di vista computazionale. Un possibile strategia consiste nell'utilizzare whitelist vuote e monitorare l'azione di un container nel tempo. In questa fase, è possibile registrare le system call necessarie per l'esecuzione. Al termine del periodo di apprendimento, sarà possibile generare delle whitelist idonee al container in esame.

Docker fornisce supporto per SECCOMP mediante BPF. Esso risulta abilitato nelle impostazioni di default ed utilizza un'ampia whitelist che esclude solo una parte delle system call del Sistema Operativo [8]. Tale lista può essere in ogni momento sovra-scritta mediante l'interfaccia utente.

Prescindendo dalla configurazione, l'impiego di SECCOMP può avere delle serie conseguenze in applicazioni critiche dal punto di vista delle prestazioni. Diversi Benchmark registrano un degrado non trascurabile [61] [62]. Ciononostante, il CIS raccomanda di non disabilitarne l'utilizzo in Docker.

2.7 Distribuzione delle immagini

Nella Sezione 2.4 è stato anticipato come l'affidabilità del software sia un elemento imprescindibile anche nel contesto della virtualizzazione leggera. La seguente sezione considera due problematiche inerenti la distribuzione delle immagini in Docker: l'utilizzo di repository come *Package Manager* e il build automatizzato delle immagini.

2.7.1 Vulnerabilità dei repository

Dal momento che la struttura del Docker Hub è simile a quella di un *Package Repository*, il Docker Daemon agisce in qualità di *Package Manager*, un meccanismo centralizzato per la gestione e l'installazione di software all'interno di un Sistema Operativo. Come tale esso è vulnerabile alla decompressione, esecuzione e salvataggio di codice non fidato [63].

Nel caso di decompressione, è possibile che un'Immagine provochi un attacco DoS durante il processo di estrazione. Difatti, dal momento che esse sono distribuite in formato compresso, una versione malevola potrebbe contenere una forma di *zip bomb*.

Un zip bomb è un file compresso creato appositamente per rendere inutilizzabile un sistema che tenta di procedere alla sua decompressione. A differenza di altri malware, esso non prevede manomissione di software. Anzi, la sua efficacia consiste nell'impossibilità di decomprimerlo da parte

di un normale programma di estrazione. Difatti, tale operazione richiederebbe enormi quantità di tempo, memoria RAM e spazio su filesystem.

Altri attacchi possono derivare dall'iniezione di codice in un'Immagine o la sostituzione di una versione aggiornata dell'immagine con una vulnerabile. Prima della versione 1.8, l'unica protezione offerta da Docker nei confronti di immagini contenenti codice malevolo era l'utilizzo di un canali TLS. Successivamente, per verificare l'attendibilità dei contenuti è stato introdotto il meccanismo di *Content Trust* (CT).

Prima di procedere all'analisi del meccanismo di Content Trust è necessario effettuare una digressione sul concetto di firma digitale.

Cifratura Asimmetrica e Firma digitale

La nascita ed evoluzione della crittografia asimmetrica deriva da un problema fondamentale presente all'interno della cifratura simmetrica: la condivisione delle chiavi. Dati due utenti A e B, si supponga di voler scambiare un documento D, da A verso B, in modo cifrato. Questo avviene mediante l'utilizzo di una chiave tra loro condivisa K. Le operazioni di cifratura alla sorgente e decifratura alla destinazione sono effettuate entrambe con K.

La cifratura asimmetrica (detta anche cifratura a chiave pubblica) supera questa limitazione mediante l'impiego di due chiavi possedute da ciascun attore, rispettivamente dette chiave pubblica e chiave privata. Come indicato dai loro aggettivi, la prima è destinata al pubblico dominio mentre la seconda è da custodire con estremo riserbo. Il comportamento delle due chiavi è complementare: un documento cifrato con la chiave privata può essere decifrato solo con la chiave pubblica e viceversa. Considerando lo stesso scenario precedente, la cifratura alla sorgente di un documento D, diretto da A verso B, avviene mediante $\mathrm{PuK_B}$, ossia la chiave pubblica di B. L'operazione di decifratura alla destinazione è fatta invece con la chiave $\mathrm{PrK_B}$, ovvero la chiave privata di B. La chiave $\mathrm{PuK_B}$ è disponibile al pubblico. Si ottiene dunque uno scambio confidenziale di dati senza condivisione di segreti tra i due attori.

La firma digitale consiste nell'applicazione di cifratura a chiave pubblica a un documento. Nello specifico, prevede di utilizzare la chiave privata dell'autore di un documento per poter cifrare il *digest* del documento stesso, ottenendo una *firma*. Il digest è il risultato di un'operazione matematica non invertibile applicata al documento, denominata *hash crittografico*. L'invio di un documento prevede dunque sia l'invio di una firma che del documento stesso. La firma digitale offre tre proprietà:

Autenticazione consente di verificare l'autore di un documento;

Integrità assicura che un documento non sia stato manomesso;

Non Ripudio l'autore del documento non può negare (a fini legali) l'autenticità e la paternità del documento stesso.

La verifica di una firma digitale avviene mediante la decifratura della firma mediante la chiave pubblica dell'autore. Nello specifico, il destinatario ricalcola il digest del documento ricevuto e ne confronta il risultato con quello ottenuto dalla decifratura della firma.

L'associazione di una chiave pubblica all'identità di un soggetto è effettuata mediante l'utilizzo di un *certificato digitale*. Il rilascio di quest'ultimo è effettuato nel contesto di una *Public Key Infrastructure* (PKI). Nello specifico, la PKI contempla la presenza di una *Registration Authority* (RA) per verificare l'identità delle entità richiedenti un certificato e una *Certification Authority* (CA) per l'emissione e la firma digitale dello stesso. La firma di un certificato avviene mediante la chiave privata della CA emittente. La fiducia in una CA è garantita dalla sua appartenenza ad una gerarchia, all'apice della quale è presente la *Root CA*, fidata implicitamente. Questa gerarchia definisce una catena di certificazione, la quale prevede che il certificato di ciascuna CA sia firmato da una CA di livello superiore. Il controllo della validità di un certificato può esser effettuato mediante l'utilizzo di una *Certificate Revocation List* (CRL) o del protocollo *Online Certificate Status Protocol* (OCSP). Una CRL è una lista di certificati revocati rilasciata periodicamente e firmata

da una CA o da un *CRL issuer* e disponibile in un repository pubblico [64]. Il protocollo OCSP [65] consente di ottenere informazioni circa la validità di un certificato esclusivamente nell'istante della richiesta, contattando un server detto *OCSP Responder*.

Docker Content Trust

Il *Docker Content Trust* (DCT) consente di effettuare con un Docker Registry operazioni di firma digitale e verifica delle immagini lato client. Esso garantsce la possibilità di utilizzare firma digitale per dati ricevuti e inviati a un Docker Registry. Tali firme consentono di verificare l'integrità e l'autore di un'Immagine [8, Content trust in Docker].

Il DCT è disabilitato di default su Docker. L'abilitazione di esso consente sia agli sviluppatori di firmare le proprie immagini sia agli utenti di fidarsi del contenuto dell'immagine in esecuzione. Inoltre, questa modalità rende invisibile all'utilizzatore finale qualsiasi contenuto non firmato e dunque non fidato.

Un'Immagine Docker è identificata dalla coppia `repository:tag`. Per ogni `tag` del `repository`, Docker calcola un digest del contenuto dell'Immagine utilizzando l'algoritmo SHA-256. È opportuno precisare che la firma si applica a livello di `tag`. Dunque, è possibile che l'immagine `torsec:latest` sia firmata e quella `torsec:2.1.15` non lo sia. La scelta di quale tag firmare è a descrizione dello sviluppatore.

L'infrastruttura di DCT si basa sull'utilizzo di tre diverse categorie di chiavi: *tagging key*, *offline key* e *server-managed key*.

Una tagging key è generata dallo sviluppatore per ogni `repository` e permette di firmare ogni `tag` appartenente al `repository`. Essa previene attacchi di tipo *Image Forgery*, ovvero contraffazione di un'Immagine. Questa può avvenire nel momento in cui, a causa di un *account hijacking* (furto di credenziali), un attaccante accede all'account di uno sviluppatore su un Docker Registry. Grazie alla presenza di una tagging key, egli non può manomettere il contenuto di un'immagine senza invalidarne la firma.

La offline key identifica uno sviluppatore o un'organizzazione. Essa è impiegata per creare le tagging key ed è custodita localmente. È generata una sola volta ed è consigliato il suo salvataggio in un dispositivo hardware non connesso alla rete, da cui il termine *offline*.

Tra le server-managed key si trova la *timestamp key*. Essa può essere utilizzata dal server per evitare attacchi di tipo *Replay*. Questo rappresenta un tipo di attacco MITM nel quale un attaccante, dopo aver intercettato un contenuto, lo ridistribuisce impersonando il legittimo autore del contenuto. Ad esempio, si considerino gli utenti A e C e il server B. La procedura di autenticazione di A presso B prevede che A invii esclusivamente il digest di un segreto condiviso con B. Se C riesce ad intercettare il pacchetto contenente tale digest può utilizzarlo successivamente per autenticarsi su B, pur non avendo alcuna conoscenza del segreto condiviso tra A e B.

Nel caso di container, un attaccante entrato in possesso di un'immagine firmata oramai vulnerabile e obsoleta, potrebbe ridistribuirla fingendosi l'autore, pur non avendo alcuna conoscenza della chiave privata utilizzata per firmarla. Pertanto, la timestamp key, la quale ha una durata limitata, prevede che un'immagine sia firmata anche da parte di un server, assicurando che allo scadere della sua validità o alla sua revoca, ogni immagine firmata con essa non sia più ritenuta valida.

2.7.2 Build automatico

Docker offre la possibilità di effettuare il build di un'Immagine automaticamente, partendo da un *build context* salvato su di un *Code Repository*, Github [66] o BitBucket [67], e collegato ad un account su Docker Hub. Nello specifico, un build context è costituito da un Dockerfile e tutti i file da esso richiamati [8, docker build].

In caso di build automatico, Docker offre la possibilità impostare dei *Webhook* [8, Webhooks for automated builds]. Essi consentono di scatenare delle azioni predefinite in risposta ad eventi

verificatisi nei succitati Code Repository, ossia il Build di una nuova Immagine o l'aggiunta di un nuovo `tag`.

Build automatico e Webhook costituiscono una pipeline in cui ogni componente ha accesso al codice destinato alla produzione.

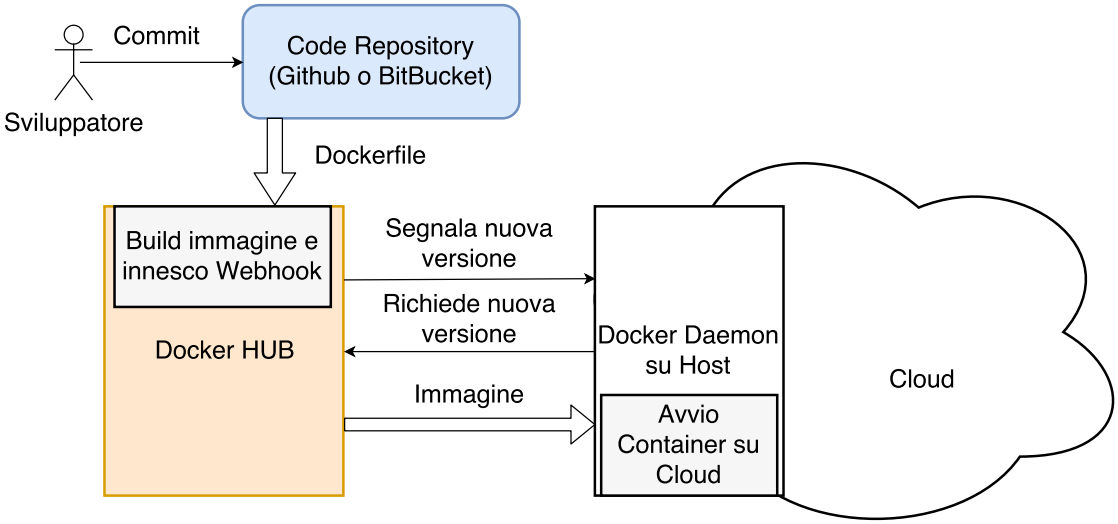

Figura 2.7. Build automatico su Docker.

La Figura 2.7 rappresenta un esempio di pipeline. Un'operazione di *commit* riguardante un Dockerfile, su Github, scatena il build automatico di un'immagine su un account Docker Hub. Da qui, può essere innescato un Webhook, il quale invia una segnalazione di aggiornamento ad un Docker Daemon in ascolto su un host raggiungibile da Internet. Quest'ultimo potrebbe scaricare la nuova immagine creata su Docker Hub e avviarla in un ambiente di testing o direttamente in produzione.

Sebbene il meccanismo appena esposto costituisca un'esemplificazione di uno scenario reale, esso evidenzia il coinvolgimento di diversi attori, ognuno dei quali con una propria superficie d'attacco. È stato già provato che, in presenza di build automatico e webhook, un *account hijacking* su un repository es. Github conduce in pochi minuti all'avvio di container compromessi in un ambiente di produzione [63]. Lo stesso risultato può essere ottenuto mediante accesso non autorizzato ad un account su Docker Hub.

L'architettura descritta non può essere protetta mediante lo schema di Content Trust descritto nella Sezione 2.7.1, dal momento che l'operazione di build non è effettuata lato client ma sul Docker Hub. È opportuno dunque che, in caso di build automatico e in mancanza di firma digitale, i container siano eseguiti in primo luogo in un ambiente di test e, una volta verificata la loro funzionalità, utilizzati in produzione.

Capitolo 3

Reverse Proxy

3.1 Overview

L'RFC-3040 [68] definisce un Reverse Proxy come intermediario, o *gateway*, autorizzato ad agire per conto e in collaborazione con uno o più *Origin Server*. Questi ultimi sono considerati detentori o creatori di una specifica risorsa.

Il Reverse Proxy si interpone tra un Origin Server e un client, occupandosi di inoltrare le richieste di quest'ultimo all'Origin Server. Dalla prospettiva del client, rappresenta il terminatore di una connessione di rete.

Il Reverse Proxy *inverte* il concetto di gateway utilizzato in un'architettura di rete con Forward Proxy. In questa, difatti, è l'Origin Server che considera il Forward Proxy come terminatore della connessione con il client. Nella Figura 3.1 sono riportate entrambe le tecnologie all'interno di uno scenario di rete. Gli Origin Server costituiscono il *backend*.

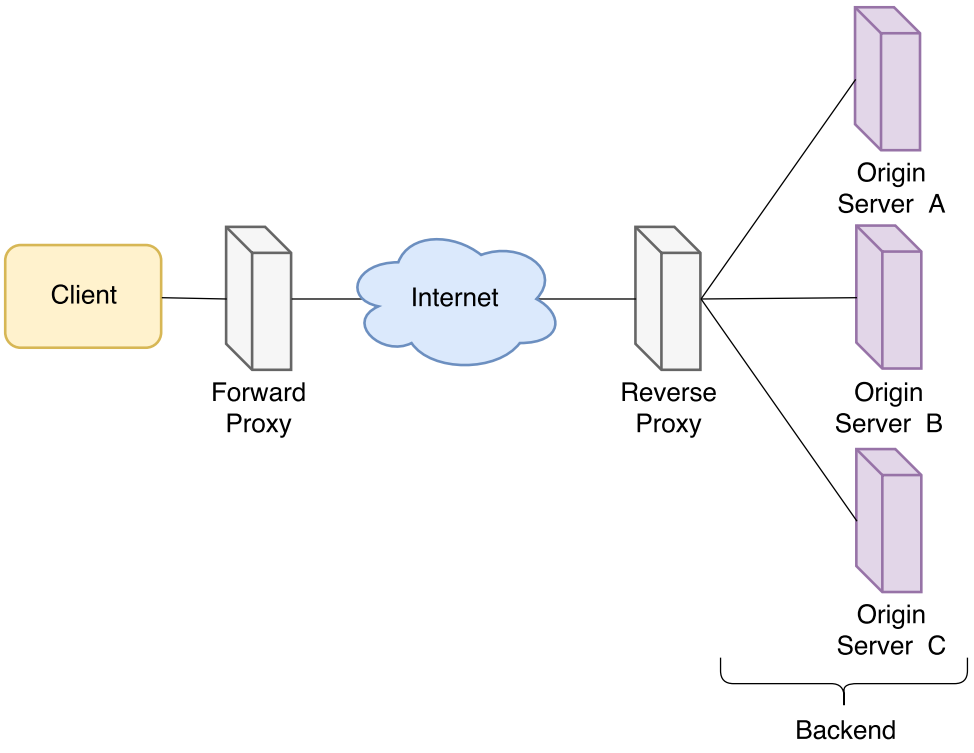

Figura 3.1. Architettura di rete con Forward Proxy e Reverse Proxy.

I vantaggi derivanti dall'introduzione di un Reverse Proxy all'interno dell'architettura di rete sono molteplici [69]. Tra questi:

Protezione. Costituisce uno strato di difesa nei confronti di un Origin Server. L'assenza di una diretta interazione tra il client e il server impedisce operazioni quali il *port scanning*. Quest'ultimo può essere utilizzato da un eventuale attaccante per rilevare la presenza di servizi attivi all'interno di un Web Server che siano obsoleti o vulnerabili. La presenza di un Reverse Proxy impedisce la raccolta di tali informazioni.

Single point of control. Effettua un controllo di accesso centralizzato. Ogni richiesta diretta verso un Origin Server può essere analizzata e, se necessario, filtrata.

Trasparenza. Il client può connettersi ad un Origin Server grazie alla sola conoscenza del nome DNS del Reverse Proxy. Eventuali cambiamenti all'interno della rete di backend sono completamente invisibili al client.

Load Balancing. Permette la suddivisione del carico tra più Origin Server. Qualora vi siano diverse richieste per un Origin Server, è possibile replicare quest'ultimo su più istanze e lasciare al Reverse Proxy il compito di ripartire le richieste in maniera opportuna tra tutte le istanze.

Spoon Feeding. In assenza di Reverse Proxy, un Origin Server trasmette dati ad un client secondo la velocità consentita dalla banda del canale che li collega. In caso di connessioni lente, un Origin Server è costretto a mantenere occupate risorse computazionali per lungo tempo esclusivamente per la trasmissione dei dati. Al fine di svincolarsi da questo incarico, un Origin Server può inviare tutti i dati destinati ad un client esclusivamente al Reverse Proxy, con il quale è instaurata una connessione TCP veloce. In questo modo, un Origin Server serve una richiesta nel tempo necessario per elaborarla ed inviarla al Reverse Proxy. Il Reverse Proxy si occupa, a sua volta, di trasmettere i dati al client secondo la velocità della connessione che li collega. In questo modo, l'Origin Server può utilizzare le risorse computazionali destinate alla trasmissione verso il client per l'espletamento di altre funzioni.

Tuttavia, l'utilizzo di un Reverse Proxy per la centralizzazione del flusso di richieste presenta degli aspetti negativi [69]:

Single Point of Failure (SPoF). Un suo malfunzionamento provoca l'isolamento totale di tutti gli Origin Server.

Sicurezza. Una vulnerabilità nel software del Reverse Proxy causa anche l'esposizione all'attaccante della rete di backend.

Prestazioni. Le operazioni di traduzione o manipolazione delle richieste effettuate dal Reverse Proxy possono avere un impatto non trascurabile sulle prestazioni di rete.

3.2 Principali soluzioni Open Source

La presente sezione considera le principali soluzioni open-source per Reverse Proxy, analizzandone sia la configurazione che l'architettura interna.

3.2.1 Apache HTTP Server con mod_proxy

Apache HTTP Server (httpd) [70] è un progetto di *The Apache Software Foundation* che mira allo sviluppo di un Web Server HTTP open-source. Rappresenta la soluzione attualmente più adottata per l'implementazione di un Web Server [71].

Sebbene nasca come Web Server, httpd è estremamente versatile e contempla la presenza di diversi moduli per l'aggiunta di altre funzionalità. Tali moduli sono aggiunti ad httpd sotto forma di *plug-in*, ovvero sono integrati direttamente all'interno del processo httpd nel Sistema Operativo.

Il modulo `mod_proxy`, con le sue estensioni, rende httpd utilizzabile come proxy/gateway. Nello specifico, esso permette la configurazione sia di un Forward Proxy che di un Reverse Proxy.

In modalità Reverse Proxy, httpd può essere integrato con il modulo `mod_cache`, il quale crea una cache di contenuti conforme alla RFC-2616 [72]. Inoltre, esso supporta diversi algoritmi di Load Balancing. Questi consentono, in caso di Origin Server replicato in più istanze, di de-modulare il flusso di connessioni dei client su più repliche.

Il design di httpd prevede diverse modalità per la gestione le richieste. In questo modo, quando utilizzato come Reverse Proxy, è possibile adattarlo sia alle esigenze di un Origin Server sia al Sistema Operativo dell'host. Attualmente sono disponibili tre soluzioni: *prefork*, *worker* e *event*.

La modalità prefork prevede l'utilizzo di un pool di processi, senza l'impiego di thread. Un processo padre regola l'allocazione di processi in tale pool, nel quale ciascuno di essi si occupa della gestione di una singola connessione client. Questa modalità consente la compatibilità con librerie di terze parti non thread-safe e permette un maggior isolamento delle richieste. Difatti, a differenza dei thread, i processi non condividono porzioni di memoria RAM tra loro, risultando maggiormente isolati.

La modalità worker implementa un modello ibrido *multi-process* e *multi-thread*, in cui ogni processo controlla un set di thread. Tale schema prevede la gestione di una connessione per thread, consentendo di servire un maggior numero di richieste e utilizzare meno risorse, al costo di un minor isolamento.

Il modello event prevede un approccio asincrono alla gestione delle richieste. Eredita dalla modalità precedente lo schema ibrido, tuttavia non prevede l'assegnazione di un thread per ogni connessione. Difatti, esso contempla l'utilizzo di un thread specifico per ogni processo che tenga traccia delle connessioni aperte, evitando che altri thread rimangano in *idle* (senza svolgere lavoro utile) durante l'attesa di richieste successive.

3.2.2 Apache Traffic Server

Traffic Server [73] è un proxy server sviluppato in origine da Inktomi e successivamente acquisito da Yahoo!. Nel 2009, il suo codice sorgente è stato donato alla Apache Software Foundation e rilasciato gratuitamente con licenza open-source Apache License 2.0 [74].

Apache Traffic Server (ATS) può essere impiegato come Reverse Proxy, Forward Proxy o cache. Esso non prevede funzionalità di Load Balancer, sebbene vi siano sperimentazioni a riguardo [75].

Analogamente ad httpd, ATS presenta una struttura modulare. Il suo core prevede la gestione delle sole funzionalità di cache e proxy mentre estensioni possono essere introdotte mediante relativi plugin. Questi ultimi si interfacciano con il core mediante un API.

Il design di Traffic Server consiste in un ambiente multi-thread asincrono. Esso non prevede l'assegnazione di un thread per ogni singola connessione, ma un modello ad eventi. I componenti che costituiscono tale architettura sono due: i *processor* e i *continuation*.

I processor si occupano di aggregare eventi in base alla loro semantica (Network I/O o cache) e di schedularne la gestione su un set di thread. Ogni thread, a sua volta, espleta il proprio compito utilizzando un oggetto continuation.

I continuation sono oggetti istanziati all'avvio di ATS o in presenza di una specifica richiesta e rappresentano delle macchine a stati. Essi ricevono in input un evento, ne elaborano il contenuto e attendono una nuova segnalazione da parte di un thread.

3.2.3 Varnish Cache

Varnish Cache [76] è un Reverse Proxy HTTP open-source. Definito dai suoi sviluppatori anche come *web accelerator*, mira ad aumentare le prestazioni di un Origin Server mediante un meccanismo di cache altamente performante. Tra le sue funzioni rientra anche quella di Load Balancer.

Varnish definisce un proprio linguaggio per le sue configurazioni, il *Varnish Configuration Language* (VCL). Esso specifica delle routine da effettuare nelle diverse fasi che costituiscono la gestione di una transazione HTTP, dall'arrivo della richiesta fino al recupero dei file dagli Origin Server. Il linguaggio VCL è compilato da Varnish Cache ed eseguito per ogni rihiesta entrante.

Il design di Varnish Cache prevede la presenza di due processi: il *manager* e il *worker*.

Il manager si occupa della compilazione dei file VCL, l'inizializzazione e la gestione del processo worker. Di questo, ne monitora lo stato mediante un meccanismo detto *heartbeat*, riavviandolo o terminandolo se necessario. Il manager offre una CLI per l'interfacciamento con l'amministratore, accessibile sia in locale che in remoto.

Il processo worker si occupa della gestione del traffico HTTP. Esso istanzia diversi tipi di thread, per il refresh della cache, la ricezione e la gestione delle richieste. Analogamente ad httpd in modalità worker, Varnish Cache assegna un thread per ogni connessione client attiva.

3.2.4 HAProxy

HAProxy [77] è un Load Balancer e Reverse Proxy open-source per applicazioni TCP e HTTP. A differenza delle soluzioni precedenti, esso si limita a riportare i contenuti degli Origin Server senza interferire con alcun meccanismo di cache.

L'architettura di HAProxy prevede l'esecuzione di un unico processo per l'intera applicazione. Questo è collocato all'interno di un *chroot jail* [79], un meccanismo di virtualizzazione che nasconde ad un software la visione dell'intero filesystem dell'host [80]. Dal punto di vista operativo, tale restrizione impedisce ad HAProxy qualsiasi cambiamento a runtime della sua configurazione o delle sue dipendenze, rendendo difficile la sua manomissione.

L'architettura di HAProxy prevede un engine con un singolo processo/thread e una coda di eventi (*event-loop*) per la gestione delle connessioni. Esso effettua la maggior parte della computazione nel kernel mediante *TCP Splicing*. Il TCP Splicing è una tecnica che mira a costituire un accoppiamento *stretto* tra le due connessioni che costituiscono un architettura di rete con proxy: la connessione diretta dall'utente al proxy e quella dal proxy all'Origin Server.

In un Sistema Operativo Linux, i dati ricevuti da un dispositivo di Input e Output, in questo caso la scheda di rete, sono salvati inizialmente nella porzione di memoria riservata al kernel, detta *kernel-space*. Un software di Reverse Proxy, al fine di poter interagire con i dati, deve prima possederli all'interno della sua porzione di memoria, denominata *user-space*. A tal proposito, è necessario che sia disposta un'operazione di copia del contenuto di ciascun pacchetto dal kernel-space allo user-space.

Il TCP Splicing riduce il numero di onerose copie di dati tra user-space e kernel-space [81] rendendo il kernel direttamente responsabile dell'invio dei dati. Difatti, HAProxy riduce al minimo indispensabile le interazioni con il contenuto dei pacchetti limitandosi a suggerire al kernel le modalità di instradamento dei pacchetti. È stimato che l'utilizzo del TCP Splicing in HAProxy consente di svolgere l'85% dell'elaborazione delle richieste nel kernel in caso di connessioni non persistenti e il 70% in caso di connessioni persistenti [78]. Una connessione non persistente prevede che il server, elaborata una singola richiesta, chiuda immediatamente la connessione con il client. Al contrario, una connessione persistente prevede che all'interno della stessa siano servite diverse richieste.

3.2.5 nginx

nginx [82] è un server HTTP e un Proxy Server open-source. Può essere utilizzato come proxy TCP/UDP, IMAP/POP3 e HTTP. Include anche funzionalità di Load Balancer e Content Cache. Attualmente rientra tra le soluzioni più utilizzate nel settore, seconda solo ad Apache HTTP Server [71].

L'architettura di nginx prevede un modello ad eventi e tre diversi tipi di processi: *master*, *worker* e *helper*.

Il processo master si occupa di effettuare operazioni che richiedono privilegi di `root`, quali il binding (collegamento) con porte con identificativo minore di 1024 (porte privilegiate) e la lettura di file di configurazione. Inoltre, si occupa di istanziare processi appartenenti alle altre categorie succitate.

I processi worker ricevono dal master una configurazione e un set di socket in ascolto. Esso serve le richieste HTTP mediante un approccio non bloccante. Tale meccanismo non attende il trasferimento di dati, ma reagisce solo in caso di segnalazione di nuovi dati disponibili all'elaborazione. Nelle impostazioni di default, nginx prevede l'assegnazione di un worker per ogni core disponibile sull'host.

I processi helper riguardano principalmente la gestione della cache. Tra questi si distinguono il *cache loader* e il *cache manager*. Il primo carica contenuti dal disco alla memoria RAM mentre il secondo si occupa del refresh della cache.

3.3 Integrazione con Web Application Firewall

Nel 2016, l'attacco ad Applicazioni Web, o *Web Application*, è risultato tra i maggiori pattern di attacco nel mondo della sicurezza dei sistemi informatici [83]. Un *Web Application Firewall* (WAF) è uno strumento per la protezione di Web Application. Può essere adoperato sia per filtrare traffico malevolo sia per operazioni di *Virtual Patching*. Esso consiste nell'applicare una mitigazione, o *remediation*, ad una vulnerabilità di una Web Application senza agire su di essa, riducendo la durata della *Window of Exposure* (WoE). Questa rappresenta l'intervallo di tempo che intercorre tra la scoperta di una vulnerabilità e l'applicazione di una relativa correzione, detta *patch*.

L'obiettivo di un WAF è quello di proteggere una Web Application, aggiungendo uno strato di difesa ulteriore a quello fornito dal Firewall o da un *Intrusion Prevention System* (IPS) [84]. Esso protegge da attacchi di livello applicativo quali *SQL Injection* (SQLi) e *Cross Site Scripting* (XSS) (Sezione 3.4).

Le organizzazioni no-profit per la sicurezza delle Web Application *Open Web Application Security Project* (OWASP) [85] e *The Web Application Security Consortium* (WASC) [86], sostengono il progetto *The Web Application Firewall Evaluation Criteria Project* (WAFEC) [87]. Quest'ultimo mira ad accrescere la comprensione di Web Application Firewall nella protezione di un Web Server e stilare una serie di criteri per aiutare gli utenti nella selezione di un WAF. Nel 2006 è stata rilasciata la versione 1.0 [88] del documento WAFEC.

3.3.1 Architetture di rete con WAF

Nella versione 1.0, il progetto WAFEC espone le possibili configurazioni di rete con WAF: Bridge, Router, Reverse Proxy ed Embedded.

La configurazioni Bridge e Router prevedono la ridirezione del traffico verso il WAF, il quale agisce in maniera trasparente come apparecchiatura di rete di livello due o tre. Sebbene efficienti, tali configurazioni sono considerate limitate dal punto di vista delle funzionalità, dal momento che non consentono di intervenire sul traffico a livello applicativo.

La modalità Reverse Proxy prevede che il WAF agisca da intermediario tra il client e il server, in modo analogo ad un Reverse Proxy. In caso di cifratura della comunicazione, essa prevede la decifrazione dei dati a livello applicativo per poter ispezionare il contenuto delle transazioni. Pertanto, può intervenire e manipolare pacchetti per fini di sicurezza es. rimozione di contenuti malevoli. La modalità Reverse Proxy richiede la riconfigurazione del DNS o la ridirezione del traffico.

La modalità Embedded prevede l'installazione di un WAF come modulo del Web Server o applicativo indipendente direttamente sul server fisico. Essa elimina lo svantaggio del Single Point of Failure nella rete introdotto nella Sezione 3.1. Tuttavia, introduce un carico computazionale maggiore sull'host del Web Server. Non richiede cambiamenti all'architettura di rete preesistente.

La letteratura disponibile offre la definizione di un'altra modalità di impiego di un WAF, non prevista nella versione 1.0 del WAFEC. Tale modalità prende il nome di *Internet Hosted/Cloud* [84].

La modalità Internet Hosted/Cloud è simile alla modalità Reverse Proxy, poiché prevede l'azione del WAF come intermediario tra client e Origin Server. Inoltre, analogamente a questa, richiede la riconfigurazione del DNS. Tuttavia, a differenza della modalità Reverse Proxy, prevede che il software WAF non sia istanziato all'interno della rete locale dell'Origin Server, ma presso un Cloud Provider. Pertanto, una richiesta diretta da un client verso un Origin Server transita prima per il WAF in esecuzione presso il Cloud Provider e successivamente è ridiretta verso l'Origin Server di destinazione . Nell'ambito client, tale tipo di servizio software, offerto in remoto, si inquadra nel modello cloud *Software as a Service* (SaaS).

3.3.2 Modelli di rilevamento

Uno dei criteri di valutazione di un WAF stabiliti dal WAFEC è la modalità di rilevamento delle minacce. Si distinguono un modello *negativo* e un modello *positivo*.

Un modello di sicurezza negativo prevede che una transazione, se non esplicitamente indicato il contrario, sia sempre concessa. Nello specifico, una transazione T può essere sempre effettuata a meno che vi sia una regola R che stabilisca il contrario. Tale modello prevede l'istituzione di una *blacklist*, costituita da regole negative, mediante due approcci [88]: *Signature Based* e *Rule Based*.

Il rilevamento Signature Based prevede il confronto del traffico con delle stringhe o espressioni regolari note. Dunque, il rilevamento di minacce si basa sulla ricerca di una corrispondenza tra il traffico in I/O e un pattern di attacco già noto. L'approccio Rule Based si basa sull'utilizzo di regole costituite da espressioni logiche, ossia contenenti operatori logici `AND` e `OR`. Esso identifica se un determinato set di azioni da parte di un utente costituisca o meno una minaccia. Ad esempio, una richiesta indirizzata ad una pagina `admin.php` unita (`and`) ad un indirizzo IP sorgente non riconosciuto come appartenente agli amministratori costituisce una minaccia.

Il modello positivo prevede di definire solo le operazioni lecite su un'applicazione, rigettandone qualsiasi altra che non superi una fase di validazione. Esso prende anche il nome di approccio *whitelist*. Il WAFEC stabilisce che il modello positivo sia più sicuro ed efficiente poiché richiede l'elaborazione di poche regole per ogni transazione. Tuttavia, esso richiede una profonda conoscenza dell'applicazione protetta e può essere di difficile manutenzione se quest'ultima cambia rapidamente.

L'approccio perseguito dal modello positivo è perlopiù *Anomaly Based*. Quest'ultimo consiste nel profilare mediante statistiche l'attività di un sistema nel tempo, al fine di poterne tracciare un comportamento ritenuto *normale*. Una deviazione da tale comportamento oltre una determinata soglia è interpretato come un attacco.

3.4 OWASP Top Ten

Il progetto *OWASP Top Ten* [89] dell'OWASP mira ad accrescere la comprensione dei principali aspetti inerenti la sicurezza delle Web Application. Esso è rivolto contemporaneamente a sviluppatori e manager, in quanto considera la sicurezza delle Web Application sia nell'ambito della programmazione del software sia nell'analisi dei fattori di rischio [90].

Il progetto OWASP Top Ten identifica i dieci maggiori fattori di rischio per la sicurezza delle Web Application. Nelle prime posizioni, a partire dalla prima, si trovano: *Injection*, *Broken Authentication and Session Management* e *Cross-Site Scripting*.

3.4.1 Injection

L'Injection è considerato il principale tipo di attacco nei confronti di una Web Application. Esso si basa sull'utilizzo di input testuali che sfruttano la sintassi di un determinato interprete. Qualsiasi

```
<?php
$id = $_POST['id'];
mysql_query("SELECT * FROM Utenti WHERE id = $id");
...
?>
```

Figura 3.2. Pagina PHP per l'esecuzione di query SQL.

sorgente di dati, sia essa esterna o interna ad un'azienda, deve essere considerata come potenziale vettore per un attacco di Injection [90].

Le conseguenze derivanti da un attacco di Injection sono diverse: perdita o manomissione di dati, DoS o controllo remoto di un host.

Un esempio di attacco di Injection è l'*SQL Injection*. Esso consiste nell'inserimento di una query SQL all'interno di un input diretto da un client verso una Web Application. Consente un accesso non autorizzato ad una base dati.

Un attacco SQL Injection può compromettere i seguenti principi di sicurezza [91]:

Confidenzialità. Esposizione di dati a cui l'attaccante non ha diritto di accedere.

Autenticazione. Accesso ad un account senza possederne le credenziali.

Autorizzazione. Modifica di permessi eventualmente salvati su una base dati.

Integrità. Inserimento, modifica o cancellazione di dati.

L'esempio di codice sorgente riportato nella Figura 3.2 rappresenta un esempio di violazione del principio di confidenzialità. La variabile `id` è inviata dall'utente mediante una richiesta POST del protocollo HTTP. Si consideri come parametro POST `id=1234`. La query SQL risultante dall'input utente è `SELECT * FROM Utenti WHERE id=1234`. Essa restituisce le informazioni associate all'identificativo inserito all'interno della tabella `Utenti`.

Si consideri ora un parametro POST `id=1234 OR 1=1`. La query SQL risultante è `SELECT * FROM Utenti WHERE id=1234 OR 1=1`. Tale query restituisce il contenuto dell'intera tabella `Utenti` presente nella base dati, dal momento che la condizione 1=1 è sempre valida.

Un requisito di sicurezza per ciascuna Web Application consiste pertanto nella validazione di input, al fine di eliminare caratteri speciali che potrebbero portare allo sfruttamento di una vulnerabilità.

3.4.2 Broken Authentication and Session Management

Gli attacchi di Broken Authentication and Session Management consistono nello sfruttamento di vulnerabilità presenti nelle funzioni per l'autenticazione e la gestione delle sessioni in una Web Application. Essi consentono all'attaccante l'impersonificazione temporanea o permanente di un altro utente [90].

Un attacco rientrante in questa categoria può portare alla compromissione di una parte o di tutti gli account registrati presso una Web Application. Se in possesso di questi, l'attaccante può operare sulla Web Application con tutti i privilegi associati ai legittimi titolari.

L'attacco di *Session Fixation* è un esempio di attacco al Session Management. Esso consiste nell'acquisizione, da parte di un attaccante, di un identificativo di sessione (ID) valido e associato all'identità di una vittima. Nella Figura 3.3 è possibile osservare i diversi passaggi di un attacco di Session Fixation [92]:

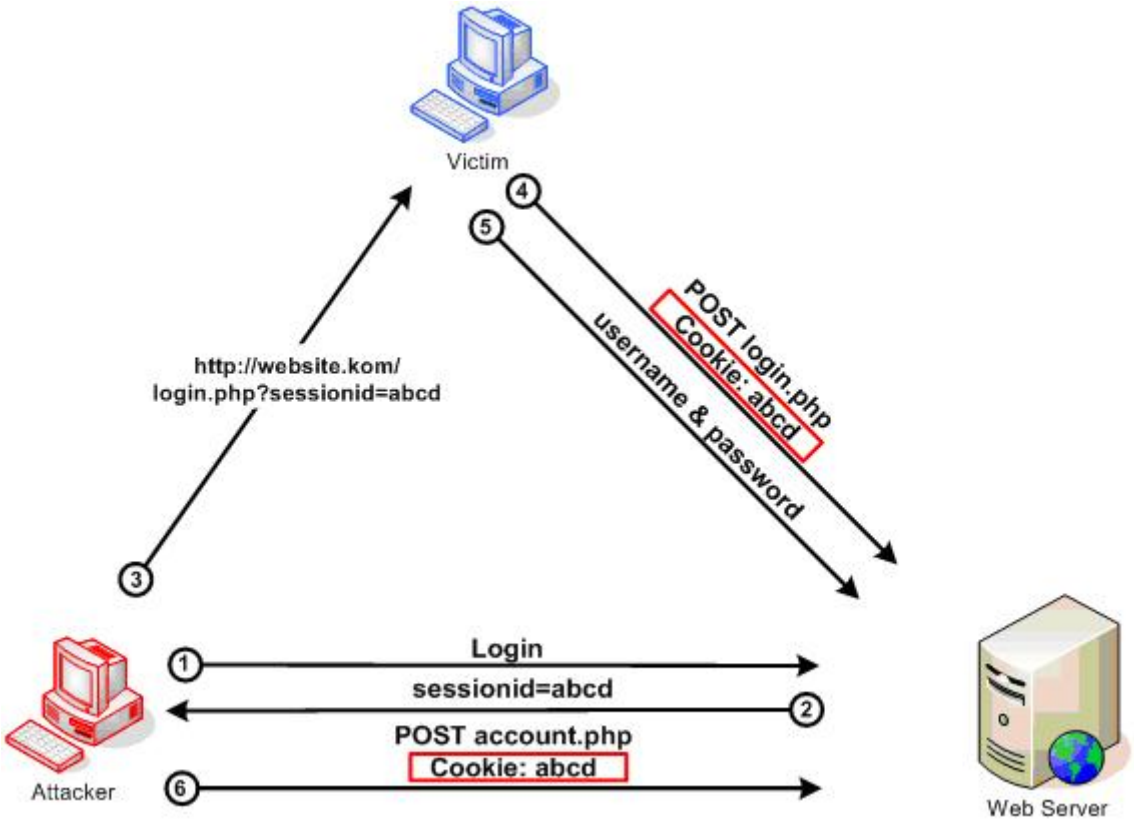

Figura 3.3. Esempio di attacco Session Fixation (fonte: owasp).

1. l'attaccante stabilisce una connessione con una Web Application;
2. la Web Application crea un apposito ID e lo restituisce all'attaccante;
3. l'attaccante invia un link con l'ID appena creato alla vittima, inducendola ad autenticarsi presso la Web Application;
4. la vittima si presenta alla Web Application con l'ID ricevuto dall'attaccante, evitando che la Web Application ne generi un altro;
5. la vittima fornisce le sue credenziali alla Web Application per autenticarsi;
6. l'attaccante può agire sulla Web Application impersonando la vittima, dal momento che l'ID generato in partenza è associato ad una sessione autenticata.

Al fine di prevenire questo tipo di attacco, l'OWASP suggerisce l'implementazione, all'interno di una Web Application, dei requisiti di sicurezza per la gestione dell'autenticazione e delle sessioni dell'*Application Security Verification Standard* (ASVS) [93]. Esso è un progetto dell'OWASP che espone una lista di test da effettuare per valutare la sicurezza di una Web Application.

3.4.3 Cross-Site Scripting

Un attacco Cross-Site Scripting (XSS) consiste nell'utilizzo di una Web Application, da parte di un attaccante, per l'invio di codice malevolo ad un utente. Tale codice si manifesta generalmente sotto forma di script es. JavaScript interpretabile da un Web Browser e può essere eseguito ogni qualvolta un input utente non è correttamente validato dalla Web Application [94].

```
<html>
    <body>
        <p>Ciao <?php echo $_GET['name'] ?></php>
    </body>
</html>
```

Figura 3.4. Pagina di benvenuto in PHP.

Analogamente agli attacchi di Injection (Sezione 3.4.1), è necessario considerare qualsiasi sorgente di dati, sia essa interna o esterna, come potenziale vettore di attacco [90].

Tra le conseguenze derivanti da attacchi XSS si possono considerare la ridirezione dell'utente verso siti fraudolenti, il furto di credenziali per l'autenticazione e il Defacement (Sezione 2.5.1).

Si consideri il codice in linguaggio PHP riportato in Figura 3.4. Esso rappresenta la pagina Web `home.php` sul dominio `example.com`, che accoglie il visitatore con una stringa di benvenuto. Tale stringa preleva il nome dell'utente da un parametro inviato mediante una richiesta GET del protocollo HTTP. Un esempio di richiesta lecita per l'utilizzo della Web Application è `http://example.com/home.php?name=MarioRossi`. Tuttavia, essa non effettua alcuna validazione sull'input dell'utente. Pertanto, anche la richiesta GET contenente il parametro `name=mallory<script>window.location.href = 'http://evilsite.com/home.php'</script>` risulta valida. Quest'ultima ha l'effetto di ri-direzionare l'utente verso una pagina di un sito malevolo. Qualora un utente fosse indotto a visitare il sito example.com con il parametro GET fraudolento diventerebbe vittima di una ridirezione verso una pagina da lui non richiesta.

Al fine di prevenire questo tipo di attacco, l'OWASP suggerisce agli sviluppatori di validare qualsiasi input utente all'interno di una Web Application dinamica.

3.5 Principali WAF open-source

Le soluzioni per Reverse Proxy trattate nella Sezione 3.2 possono essere integrate con dei WAF al fine di introdurre un livello di protezione degli Origin Server indipendente da questi. In questo modo, il Reverse Proxy può procedere al filtraggio di potenziali minacce prima che esse possano introdursi nella rete di backend.

Nella presente sezione si analizzano le funzionalità e la struttura di due soluzioni open-source di WAF: ModSecurity [95] e NAXSI [96].

3.5.1 ModSecurity

ModSecurity è un WAF open-source rilasciato nella sua prima versione nel Novembre 2002. Si presenta come un toolkit per controllo di accessi, monitoraggio e log in tempo reale. Per questo motivo, è anche denominato *HTTP intrusion detection tool* [97].

ModSecurity può essere adoperato sia in modalità Embedded (Sezione 3.3.1) sia in modalità Reverse Proxy.

La prima soluzione prevede che l'architettura di rete preesistente rimanga immutata. Ad esempio, non sono introdotti SPoF. Tuttavia, la modalità Embedded richiede che Origin Server e WAF condividano le stesse risorse computazionali poiché coesistenti sullo stesso host.

La seconda modalità integra ModSecurity con un Reverse Proxy, lasciando immutato l'host su cui è in esecuzione l'Origin Server. A differenza del caso precedente, si ha un aggiornamento nella configurazione di rete che prevede la ridirezione di tutte le richieste dirette all'Origin Server verso il WAF.

Le funzionalità di ModSecurity possono essere ricondotte a quattro categorie [97]:

Parsing. Estrarre bit utili dai pacchetti per l'applicazione di regole.

Buffering. Incamerare l'intera richiesta prima di passarla all'Origin Server per poterla analizzare.

Logging. Tracciare l'intera transazione HTTP.

Rule Engine. Applicare regole sulla base delle azioni compiute dalle categorie precedenti.

L'analisi del traffico con ModSecurity avviene mediante un modello negativo (Sezione 3.3.2) e rule-based. Queste sono utilizzate sia per identificare potenziali minacce sia per le operazioni di log. Una regola specifica una determinata espressione e quale azione intraprendere in caso di un riscontro all'interno di una transazione. L'ispezione di una transazione si articola in cinque *fasi* differenti, distinte in base all'entità su cui la regola è applicata:

1. Request Header;
2. Request Body;
3. Response Header;
4. Response Body;
5. Log.

La terminazione di ogni fase è propedeutica alla successiva, in modo tale da bloccare eventuali minacce prima di procedere ulteriormente all'elaborazione della richiesta. Ad esempio, l'ispezione del Request Body è effettuata solo se l'analisi del Request Header non abbia riscontrato anomalie. In questo caso, si evita di procedere all'elaborazione del corpo della richiesta se non necessario [97].

ModSecurity può essere integrato con il progetto *OWASP ModSecurity CRS* [95, CRS], che propone un set di regole per il rilevamento di un ampio numero di attacchi, includendo quelli facenti parte della *OWASP Top Ten*. A titolo d'esempio si possono considerare SQLInjection e XSS.

ModSecurity può essere integrato in modalità Reverse Proxy con httpd o nginx.

3.5.2 NAXSI

NAXSI è un progetto open-source per lo sviluppo di un WAF compatibile con il server nginx. Esso si propone per contrastare diversi tipi di attacchi, tra cui quelli rientranti nella *OWASP Top Ten*, mediante un meccanismo basato su whitelist [98].

Il funzionamento di NAXSI prevede l'utilizzo di regole che valutino la presenza di caratteri ed espressioni potenzialmente dannosi all'interno di una transazione. Ad esempio, i caratteri `<>'(){}`. Nello specifico, NAXSI assegna un indice di *pericolosità* ad ognuno dei caratteri succitati. La presenza di ciascuno di essi incrementa un *punteggio* globale per una richiesta. Qualora il punteggio eccedesse una soglia predefinita dall'utente, il WAF intraprenderebbe una determinata azione.

L'approccio appena descritto si dimostra efficace in caso di vulnerabilità non note, indipendentemente dal linguaggio di programmazione utilizzato. Dal momento che impiega un modello di rilevamento positivo (Sezione 3.3.2), NAXSI richiede una conoscenza preliminare dell'applicazione che si intende proteggere. Pertanto, prevede due modalità di funzionamento: *learning mode* e *production mode*.

Nella fase di learning, tutte le richieste ritenute pericolose sono riportate nei file di log, ma non bloccate. Al fine di creare un modello è necessario che la tal fase abbia una durata dipendente dalla dimensione dell'applicazione web e dalla complessità degli input [99]. Una volta terminata la fase di learning, è possibile generare, a partire dai file di log delle whitelist, mediante il tool `nxapi`. Queste indicano a NAXSI quali pattern di input non bloccare al fine di non generare falsi positivi.

In production mode, NAXSI blocca e riporta nei file di log le richieste ritenute pericolose sulla base dei risultati dello step precedente. Tuttavia, è possibile che transizioni successive evidenzino

ulteriori falsi positivi. Pertanto, NAXSI offre la possibilità di rigenerare whitelist in un processo ciclico a partire dai file di log, mediante `nxapi`.

La Figura 3.5 mostra come durante la fase di learning sia possibile integrare con NAXSI strumenti per il salvataggio, l'analisi e la visualizzazione di dati come Elasticsearch [100] e Kibana [101]. L'interazione con essi avviene mediante `nxapi`.

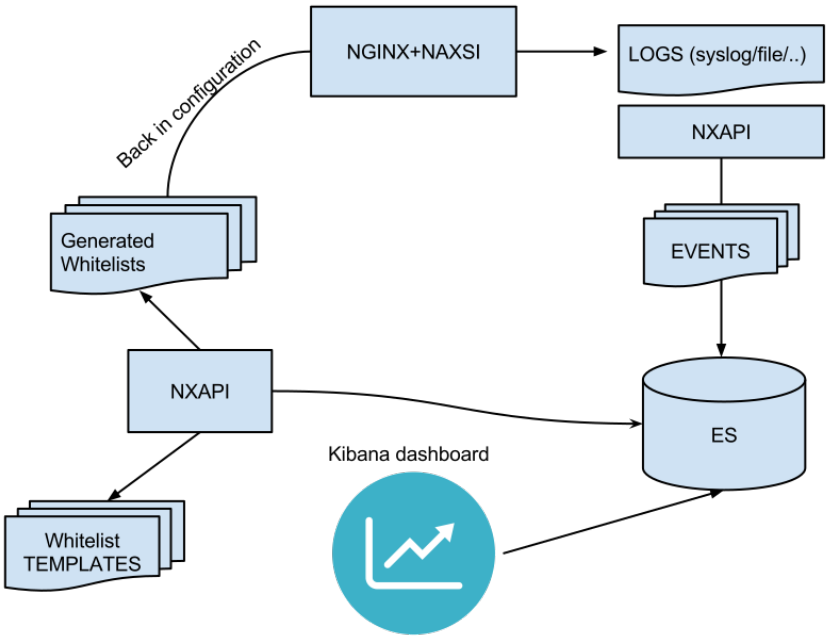

Figura 3.5. Workflow di NAXSI (fonte: github).

3.6 Ambiente di test

Al fine di valutare l'integrazione di Reverse Proxy con WAF all'interno di una funzione di rete virtualizzata sono stati condotti diversi test, volti all'analisi del consumo delle risorse e delle prestazioni. Tali test sono stati svolti su un singolo host in ambiente Docker e non in ambiente distribuito. Laddove non sia specificato differentemente, le configurazioni citate sono costituite da soli container Docker.

3.6.1 Architettura Hardware e Software

L'ambiente di test è rappresentato da una piattaforma con la seguente architettura hardware e software:

- Intel Core i7-4510U CPU @ 2.00GHz, 2 core, 2 thread per core;
- 12 GB di RAM DDR3
- OS Debian 9 Stretch 64 bit con kernel Linux 4.9.0-3-amd64
- Docker versione 17.06.0-ce.

3.6.2 Software di benchmark

`ab` [102] è uno strumento utilizzato per la valutazione delle performance di Web Server. Nell'ambiente sperimentale che si sta definendo, esso può essere utilizzato per simulare un insieme di utenti che genera un flusso di richieste HTTP nei confronti di configurazioni sotto test. Possiede diverse impostazioni, tra cui la possibilità di definire il numero di utenti contemporaneamente attivi (concorrenza) con connessioni persistenti.

I benchmark effettuati da `ab` possono essere arrestati con diversi criteri, tra cui il raggiungimento di un numero totale di richieste o la scadenza di un timeout. In questo modo è possibile ricreare esattamente lo stesso ambiente di test con configurazioni differenti.

3.6.3 Considerazioni Preliminari

I risultati riportati sono dimensionati all'architettura descritta nella Sezione 3.6.1 e non mirano alla proclamazione della soluzione più performante o computazionalmente meno dispendiosa. Si preferisce, invece, delineare una soluzione adatta allo scenario vNSF descritto nel Capitolo 1, versatile e facilmente implementabile.

Dal momento che si sta operando in ambiente locale e non distribuito, è necessario assicurarsi che l'architettura hardware non degradi la qualità dei risultati di test. Nello specifico, le prestazioni di un singolo container su un host non sono paragonabili a quelle di un container in esecuzione sullo stesso host con altre istanze. Nel secondo caso Docker provvede di default a distribuire le risorse computazionali equamente tra ogni singola istanza. Al fine di evitare che l'architettura hardware della piattaforma di test influenzi i risultati, è stata utilizzata la direttiva di Docker `--cpuset-cpus`. Essa consente di forzare la schedulazione dei processi all'interno di container solo su determinati core della CPU. In questo modo, è possibile dividere uniformemente nei diversi casi di studio le risorse computazionali.

Infine, dal momento che si mira a valutare localmente soluzioni di Reverse Proxy con WAF è necessario assicurarsi che le prestazioni dell'Origin Server non influenzino in alcun modo i risultati dei test. Pertanto, è opportuno effettuare un test preliminare.

Test Preliminare

Il test preliminare mira a dimostrare che un Origin Server esposto direttamente sulla rete dell'host, con un numero *limitato* di utenti concorrenti e in un intervallo di tempo predefinito, completi un numero di richieste HTTP superiore a quello che completerebbe con l'intermediazione di un Reverse Proxy. In caso di successo, si può stabilire che, con gli stessi parametri di test e le stesse capacità computazionali, l'Origin Server non può provocare un degrado delle prestazioni nello scenario testato. Ad esempio, un Origin Server che soddisfi 500 richieste in 30 secondi senza l'intermediazione di un Reverse Proxy e 200 con quest'ultimo non costituisce con buona probabilità un *collo di bottiglia* per le prestazioni. Al contrario qualora vi siano 500 richieste soddisfatte in 30 secondi con o senza un Reverse Proxy è estremamente probabile che l'Origin Server costituisca un collo di bottiglia.

Il test preliminare consiste nel valutare tre diversi scenari:

- Origin Server senza Reverse Proxy (standalone);
- Origin Server con Reverse Proxy Apache HTTP Server (httpd) senza WAF;
- Origin Server con Reverse Proxy nginx senza WAF.

In ogni configurazione le funzioni di cache non sono abilitate.

Nello scenario Origin Server standalone, l'Origin Server è direttamente raggiungibile sull'interfaccia `localhost` dell'host. Questo è possibile grazie ad un *port mapping* tra il container in cui è in

esecuzione l'Origin Server e l'host stesso. Mediante la direttiva `--cpusets-cpu` è stata vincolata l'esecuzione dell'Origin Server ad un singolo core.

La configurazione con Apache HTTP Server prevede che l'Origin Server sia raggiungibile solo attraverso il Reverse Proxy. Quest'ultimo, è esposto mediante port mapping sull'interfaccia `localhost` dell'host. Come introdotto nella Sezione 3.2.1 Apache HTTP Server può soddisfare le richieste mediante diverse modalità di elaborazione. In questo caso è stato selezionata la modalità event, ritenuta più efficiente rispetto alle precedenti [103]. La direttiva `--cpusets-cpu` è stata impiegata per assegnare un singolo core *virtuale* all'Origin Server e due core *virtuali* al Reverse Proxy.

Analogamente al precedente, lo scenario con nginx prevede che l'Origin Server sia raggiungile solo mediante il Reverse Proxy, dal momento che non vi sono port mapping tra Origin Server e host. Si è specificato un singolo core per l'Origin Server e due core per il Reverse Proxy.

L'Origin Server utilizzato in tutti e tre gli scenari è costituito da un'istanza di httpd 2.4.6 in modalità Web Server, il quale serve una solo pagina statica `index.html` della dimensione di 32 KiB.

L'assenza di un WAF evita l'introduzione di ulteriore overhead, non necessario in un questo tipo di test. Le prestazioni di un Reverse Proxy con WAF sono analizzate nella Sezione 3.8.

Ciascuna configurazione riceve un flusso di richieste `HTTP/1.1` sull'interfaccia di rete `localhost` da parte di `ab`. La durata di ogni singola esecuzione del benchmark è di 45 secondi, con una simulazione di 1000 connessioni persistenti. Al fine di ridurre la variabilità statistica, ogni esecuzione è stata ripetuta per un numero di volte pari a 3.

Nella Figura 3.6 è possibile osservare che il numero di richieste soddisfatte interagendo direttamente con l'Origin Server è superiore agli altri scenari, dimostrando il calo di prestazioni dovuto alla presenza di Reverse Proxy introdotto nella Sezione 3.1.

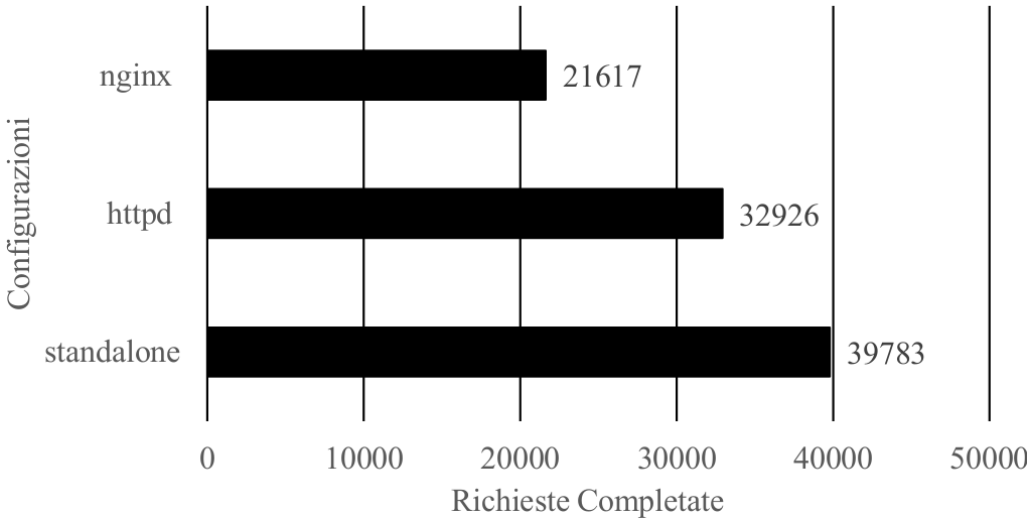

Figura 3.6. Richieste servite da Web Server di test in presenza di Reverse Proxy ed in modalità standalone.

A partire da questo risultato è possibile asserire che con un numero di utenti simulati pari a 1000, in un benchmark di 45 secondi, l'Origin Server considerato non rappresenta un collo di bottiglia. Tale configurazione rappresenta un *upper bound* per i test successivi.

3.7 Analisi utilizzo risorse

La presente Sezione mira ad analizzare il consumo di risorse da parte di differenti configurazioni software. Per ragioni di compatibilità sono state considerate le seguenti soluzioni Reverse Proxy con WAF:

- Apache HTTP Server con `modproxy` e ModSecurity (httpd + ModSecurity);
- nginx con NAXSI (nginx + NAXSI);
- nginx con ModSecurity (nginx + ModSecurity).

L'architettura di rete considerata prevede:

- 1 container Docker in cui è in esecuzione un Origin Server;
- 1 container Docker in cui sono istanziati il Reverse Proxy con WAF in esame;
- 1 processo che esegue un'istanza di `ab` direttamente sull'host.

In ogni configurazione le funzioni di cache non sono abilitate.

L'Origin Server utilizzato in tutti gli scenari è costituito da un'istanza di httpd 2.4.6 in modalità Web Server, il quale serve una solo pagina statica `index.html` della dimensione di 32 KiB.

3.7.1 Benchmark CPU share

Per l'analisi del CPU share sono stati simulati con `ab` flussi di richieste `HTTP/1.1` con header `connection=keep-alive`, come suggerito dal WAFEC [88]. Sono stati avviati benchmark della durata di 45 secondi con concorrenza pari a 250, 500 e 1000 utenti (upper bound definito nella Sezione 3.6.3). Al fine di ridurre la variabilità statistica, ogni possibile combinazione di parametri è stata eseguita tre volte.

Nella Figura 3.7 è stato profilato l'andamento del CPU share con 1000 utenti concorrenti. È possibile osservare che la combinazione httpd + ModSecurity si dimostra la più pesante dal punto di vista computazionale. Tuttavia, sia nginx + ModSecurity sia httpd + ModSecurity rappresentano una soluzione stabile dal punto di vista dell'utilizzo di risorse.

La configurazione nginx + NAXSI si dimostra la più leggera nei consumi, sebbene sia anche la soluzione con un andamento meno predicibile. Difatti, si osserva una variazione del 50% del CPU share tra il 20" e il 35" del test.

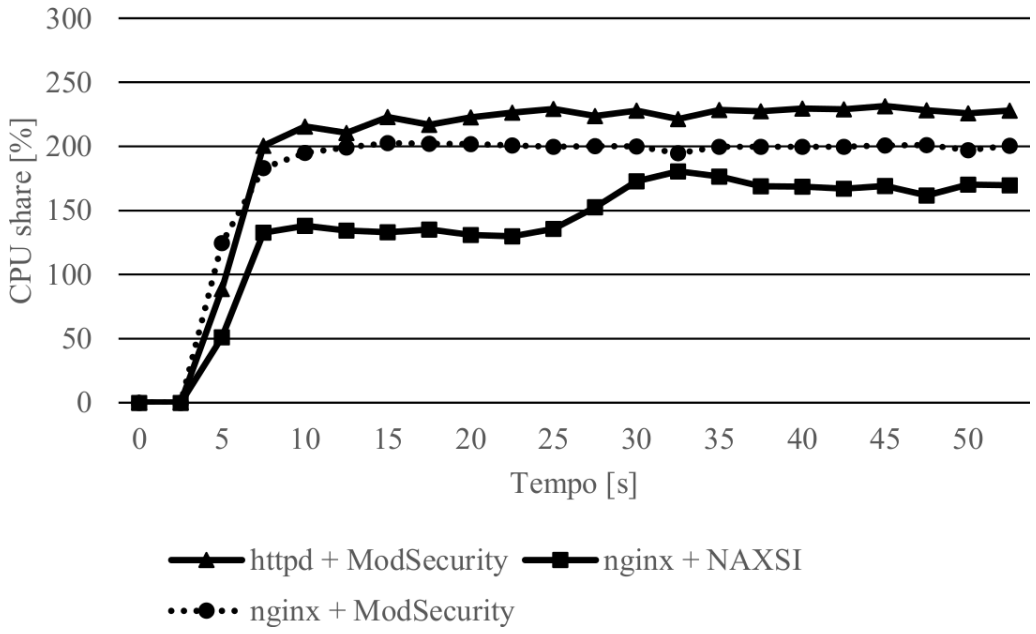

Figura 3.7. Trend consumo CPU share con numero utenti pari a 1000.

In Figura 3.8 è possibile osservare come tutte e tre le combinazioni non risentano dell'aumento di connessioni attive da 250 a 1000. Tale risultato può essere spiegato dall'architettura software in utilizzo. Difatti, tutte e tre le combinazioni di Reverse Proxy con WAF utilizzano un modello basato su eventi. Tale approccio consente di gestire le connessioni in maniera asincrona, evitando l'apertura di un nuovo thread per ogni connessione attiva.

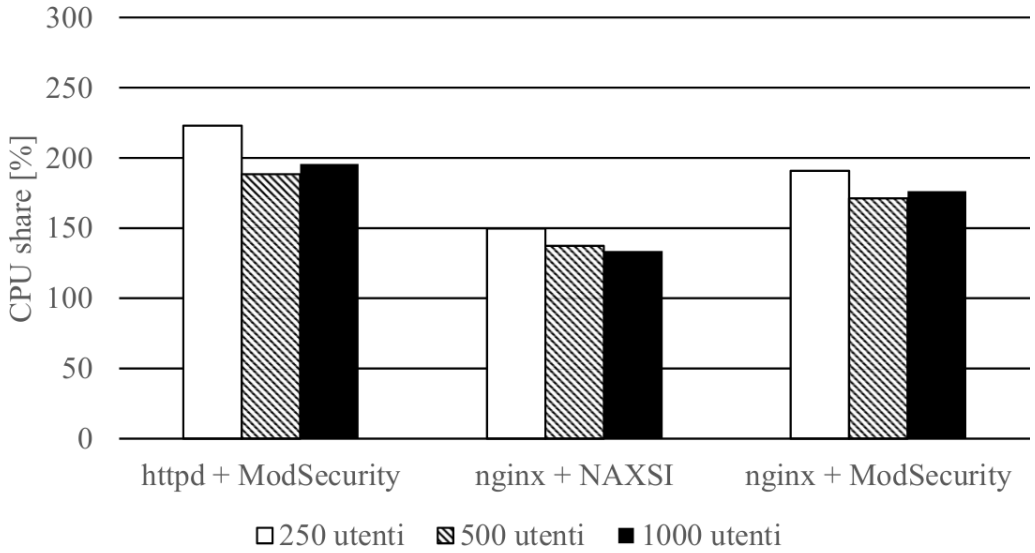

Figura 3.8. Utilizzo medio di CPU per diverse soluzioni di Reverse Proxy con WAF con numero variabile di utenti.

3.7.2 Benchmark RAM

Per l'analisi del consumo di memoria RAM sono stati simulati con `ab` flussi di richieste `HTTP/1.1` con header `connection=keep-alive`. Sono stati avviati benchmark della durata di 45 secondi con concorrenza pari a 250, 500 e 1000 utenti. Analogamente al benchmark precedente, ogni possibile combinazione di parametri è stata testata in tre esecuzioni differenti.

Nella Figura 3.9 è possibile osservare come la soluzione composta da nginx + ModSecurity si dimostri la più dispendiosa nell'utilizzo di RAM, mentre la soluzione composta da nginx + NAXSI la più leggera. Tuttavia, entrambi hanno un andamento stabile dei consumi durante la durata del test. Al contrario, il consumo di memoria nella configurazione httpd + ModSecurity ha un andamento crescente.

Nella Figura 3.10 si osserva come httpd + ModSecurity sia la configurazione più stabile nel consumo medio di RAM al crescere del numero di utenti. Al contrario, le soluzioni nginx + NAXSI e nginx + ModSecurity mostrano un trend in crescita. In particolare, nel caso di nginx + ModSecurity si osserva un aumento di circa il 35% tra i benchmark con 250 utenti concorrenti e quelli con 500. Tale incremento diviene del 50% tra la configurazione con grado di concorrenza 500 e quella pari a 1000.

3.8 Analisi di throughput

La presente sezione mira ad analizzare il valore di throughput di diverse configurazioni software. Esso è calcolato da `ab` considerando il numero di byte trasferiti nell'intervallo di tempo che intercorre tra l'instaurazione della prima connessione e la chiusura dell'ultima, in questo caso è pari a 45 s.

Analogamente alla Sezione 3.7, sono state considerate le seguenti soluzioni Reverse Proxy con WAF:

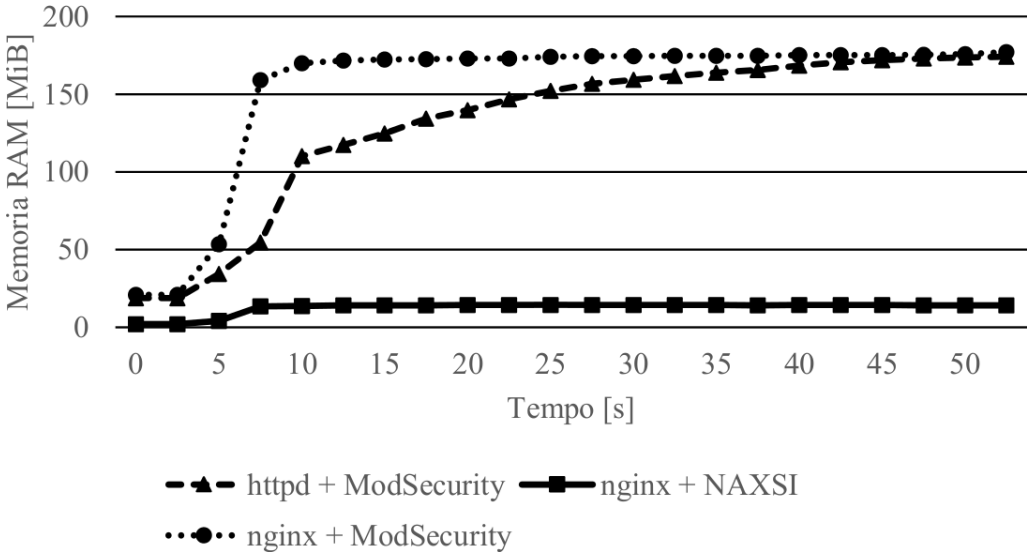

Figura 3.9. Trend consumo memoria RAM con numero utenti pari a 1000.

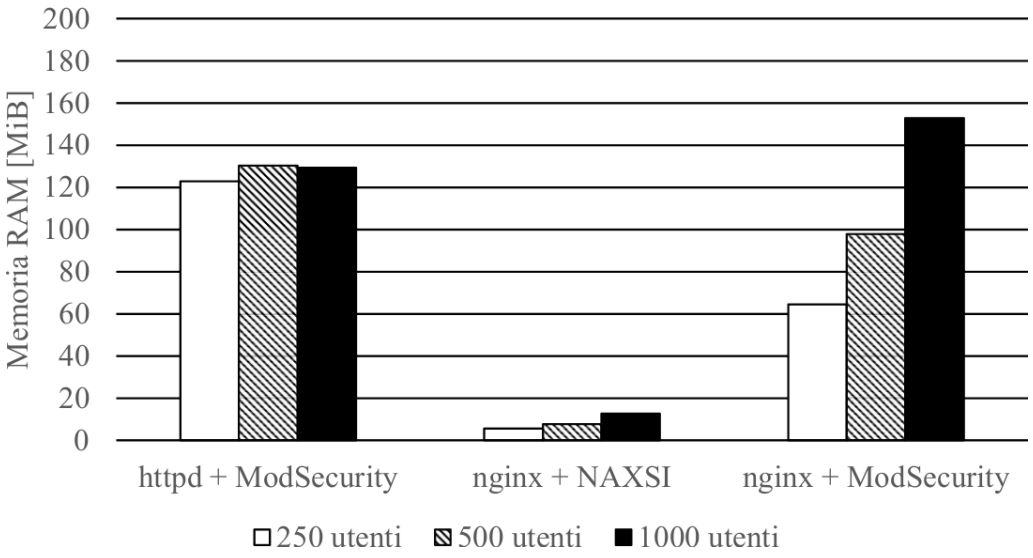

Figura 3.10. Utilizzo medio di RAM per diverse soluzioni di Reverse Proxy con WAF con numero variabile di utenti.

- Apache HTTP Server con `modproxy` e ModSecurity (httpd + ModSecurity);
- nginx con NAXSI (nginx + NAXSI);
- nginx con ModSecurity (nginx + ModSecurity).

Inoltre, l'architettura considerata prevede:

- 1 container Docker in cui è in esecuzione un Origin Server;
- 1 container Docker contenente il Reverse Proxy con WAF in esame;
- 1 processo che esegue un'istanza di `ab`.

In ogni configurazione le funzioni di cache non sono abilitate.

L'Origin Server utilizzato in tutti gli scenari è costituito da un'istanza di httpd 2.4.6 in modalità Web Server, il quale serve una solo pagina statica `index.html` della dimensione di 32 KiB.

Nella Figura 3.11 sono messi a confronto i throughput delle diverse soluzioni in esame con una soluzione standalone. In primo luogo, emerge come le prestazioni della soluzione priva di Reverse Proxy siano di gran lunga maggiori di quelle dei concorrenti. Inoltre, è possibile osservare come nginx + NAXSI rappresenti soluzione più performante dal punto di vista delle prestazioni, con un throughput del 60% maggiore di httpd con ModSecurity. Tale risultato può essere giustificato dall'impiego di un modello di rilevamento positivo (Sezione 3.3.2). Lo stesso nginx, tuttavia, ha prestazioni significativamente peggiori se utilizzato congiuntamente a ModSecurity. Si evidenzia dunque un grado di immaturità nell'attuale integrazione tra questi due.

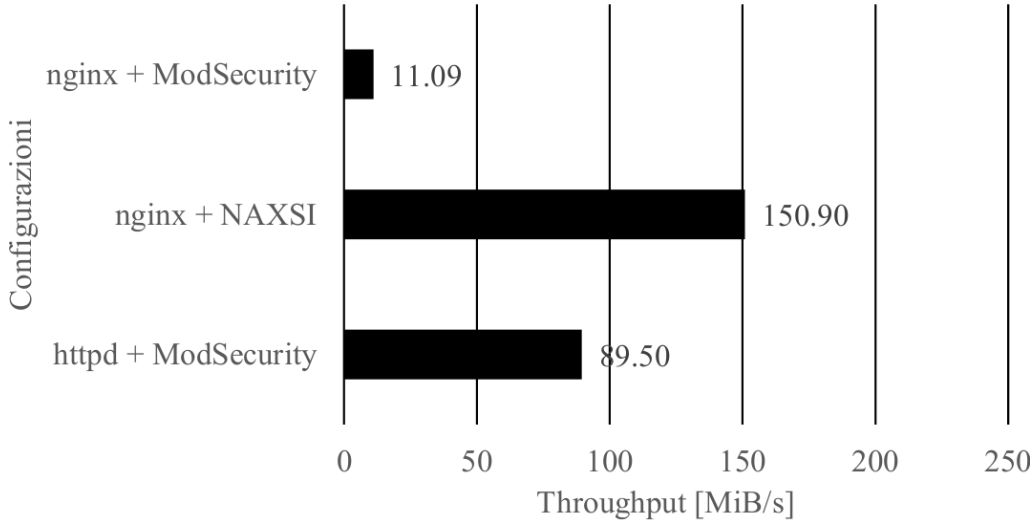

Figura 3.11. Throughput delle soluzioni Reverse Proxy con WAF.

3.9 Considerazioni

I risultati dei test condotti nella Sezione 3.7 e Sezione 3.8 dimostrano che nginx + NAXSI rappresenta la soluzione più performante dal punto di vista delle prestazioni e dei consumi di risorse. Tuttavia, i test non considerano il tempo di apprendimento necessario a NAXSI per essere utilizzato in production mode. Tale intervallo non è facilmente predicibile, poiché dipende dalla dimensione e della complessità degli input della Web Application che si intende proteggere.

httpd + ModSecurity si dimostra come alternativa meno valida dal punto di vista delle prestazioni rispetto a nginx e il suo consumo di risorse è superiore alle soluzioni alternative. Tuttavia, a differenza di nginx, httpd + ModSecurity può beneficiare della presenza di regole di rilevamento già definite. Dunque, non necessitando di un iniziale periodo di apprendimento può essere utilizzato immediatamente in produzione per la protezione di un Origin Server.

Sebbene anch'esso goda dei benefici di un modello di rilevamento negativo, nginx + ModSecurity si dimostra globalmente inefficiente, sia dal punto di vista delle prestazioni che dal punto di vista del consumo delle risorse.

Come introdotto nel Capitolo 1, scopo di questa tesi è la valutazione di una combinazione Reverse Proxy con WAF idonea all'implementazione in una vNSF. Per tale scenario, si richiede l'utilizzo di una soluzione software che sia immediatamente utilizzabile per la mitigazione di una vulnerabilità. Benché dimostratosi più efficiente, un modello di sicurezza positivo (Sezione 3.3.2), rappresentato in questo caso da NAXSI, non può essere applicato immediatamente all'interno di un'architettura di rete. Difatti, l'impiego di NAXSI senza la preliminare fase di learning (Sezione 3.5.2) potrebbe portare a risultati impredicibili (falsi positivi). Tale rischio è aggravato dalla

presenza di Origin Server con scopi diversi nella rete di backend. Pertanto, si è scelta l'adozione di un modello di sicurezza negativo. Dal momento che la combinazione httpd + ModSecurity si è dimostrata più efficiente nella Sezione 3.7 e Sezione 3.8 della configurazione nginx + ModSecurity, l'implementazione proposta della vNSF nel Capitolo 5 si baserà sull'utilizzo di httpd + ModSecurity.

Capitolo 4

Architettura

4.1 Overview

Nella Sezione 2.2 è stato analizzato il concetto di container, considerando sia i principi di funzionamento legati al kernel Linux sia le maggiori implementazioni presenti sul mercato, con particolare attenzione verso Docker. Il presente capitolo si propone di sfruttare tali conoscenze per analizzare i molteplici aspetti architetturali di una vNSF e dell'ambiente in cui questa è eseguita. A tal proposito, si anticipa che la trattazione è articolata in quattro diversi punti.

Inizialmente, si esamina un possibile caso d'uso delle vNSF. Esso costituisce un punto di partenza per poter delineare i principali elementi strutturali di una vNSF ed è più volte richiamato nel corso dell'esposizione.

Successivamente, si contestualizza il concetto di container nell'ambito di una vNSF, spostando l'attenzione dalla struttura del singolo alle interazioni che avvengono tra più entità. Difatti, si anticipa che una vNSF prevede la coesistenza di più container che si interconnettono. In questa parte, si delineano gli elementi di una vNSF generica ed un'applicazione concreta dell'architettura ad un caso di studio: la vNSF Reverse Proxy. Per la progettazione di quest'ultima, si riprendono i concetti esposti nel corso del Capitolo 3.

In seguito, si valutano gli aspetti che descrivono una vNSF come software, o insieme di software, in esecuzione su una piattaforma di calcolo. Tra questi rientra, ad esempio, la configurazione dei container che la compongono e dell'host su cui essi sono avviati.

Infine, si introducono alcuni criteri da rispettare in fase di progettazione per poter impiegare una qualsiasi vNSF in un ambiente di produzione distribuito.

Ad eccezione di alcuni aspetti, opportunamente segnalati, l'architettura proposta nei punti succitati si basa sull'utilizzo dei soli componenti dalla piattaforma di virtualizzazione leggera Docker, tra cui le Immagini e i Volumi, descritti precedentemente nella Sezione 2.3.

4.2 Caso d'uso: Security-as-a-Service

L'*European Telecommunications Standards Institute* (ETSI) considera il concetto di vNSF come fondamento di una *NFV Infrastructure* (NFVI) che si propone di fornire un servizio di tipo *Security as a Service* (SecaaS). Mediante virtualizzazione, è possibile sollevare l'utente dalla necessità di procurarsi apparecchiature fisiche dedicate a funzioni di sicurezza. Gli obiettivi che l'utilizzo di vNSF si pone possono essere sintetizzati nei seguenti punti [2, Security as a Service]:

- bloccare qualsiasi tipo di minaccia che può essere contrastata mediante misure sulla rete;
- garantire scalabilità che si estenda oltre le capacità della singola impresa;

- monitorare e raccogliere dati in determinati punti della rete per analisi o per applicare remediation mirate.

In Figura 4.1 è rappresentata l'interazione tra diverse entità che caratterizzano un modello SecaaS basato su NFV [2, Security as a Service]. Il processo parte dalla pubblicazione di una vNSF da parte di un'entità terza es. ISP all'interno del vNSF Store (punto 1). Esso è un catalogo centralizzato all'interno del quale si può ricercare, selezionare e commercializzare vNSF.

I punti 2 e 3 vedono l'analisi da parte del cliente dei possibili servizi offerti, interagendo con un'interfaccia denominata *dashboard*.

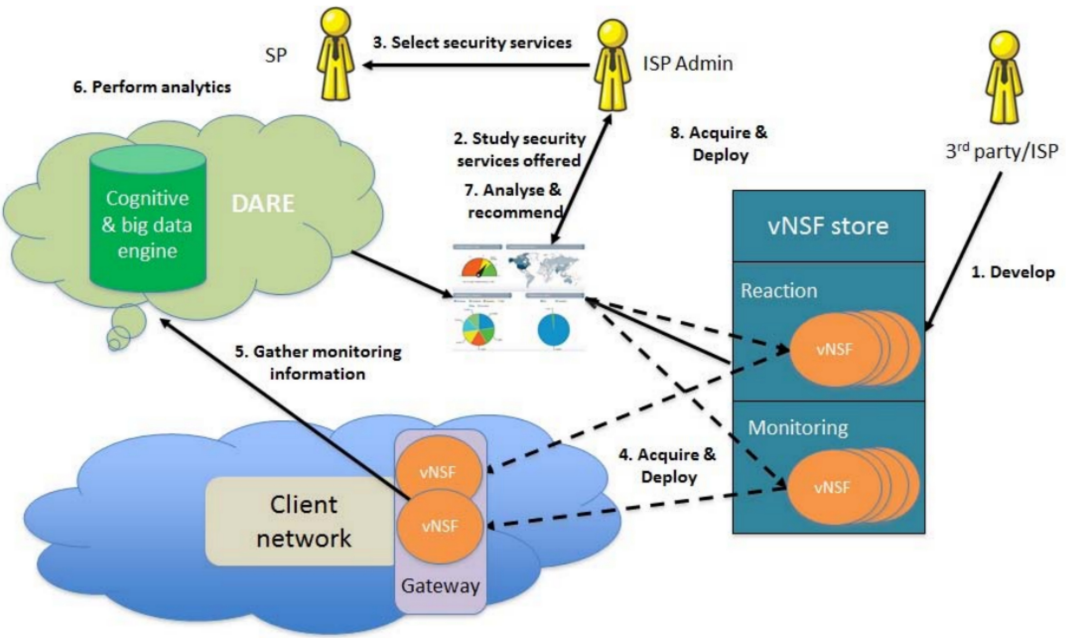

Figura 4.1. Modello SecaaS basato su NFV (fonte: ETSI [2]).

Una volta selezionata la vNSF richiesta, è possibile eseguirne il *deployment*, ossia l'avvio all'interno della NFVI (punto 4). Gli elementi che costituiscono l'infrastruttura di deployment delle vNSF prendono il nome di *NFVI Point of Presence* (NFVI PoP) mentre l'entità che si occupa del coordinamento delle policy di sicurezza si chiama *vNSF Orchestrator* (vNSFO).

Nei punti 5 e 6 si nota la partecipazione di un *Data Analysis and Remediation Engine* (DARE), il quale raccoglie e analizza dati alla ricerca di *pattern* che identifichino o prevedano comportamenti malevoli.

In seguito alle minacce analizzate e ai requisiti del client, il DARE può proporre ulteriori azioni di remediation da effettuare per la protezione della rete. A partire da questo si può partire al deployment di nuove vNSF fino a raggiungere il livello di sicurezza desiderato dall'utente.

Un'implementazione di tale architettura è presente all'interno del progetto europeo SHIELD [104], il quale propone un'infrastruttura per il deployment di vNSF all'interno delle reti di ISP o reti aziendali.

4.3 Architettura di una vNSF

Nella Sezione 2.1 sono stati elencati i principali punti di distacco tra Virtualizzazione Hardware e Virtualizzazione Leggera. Pertanto, come dimostrato in seguito, le scelte architetturali dipendono dalla tecnologia adoperata.

4.3.1 Struttura generica di una vNSF

L'architettura di una vNSF proposta nel presente lavoro di tesi prevede la coesistenza e la collaborazione tra più entità. La Figura 4.2 si propone di evidenziare tale aspetto.

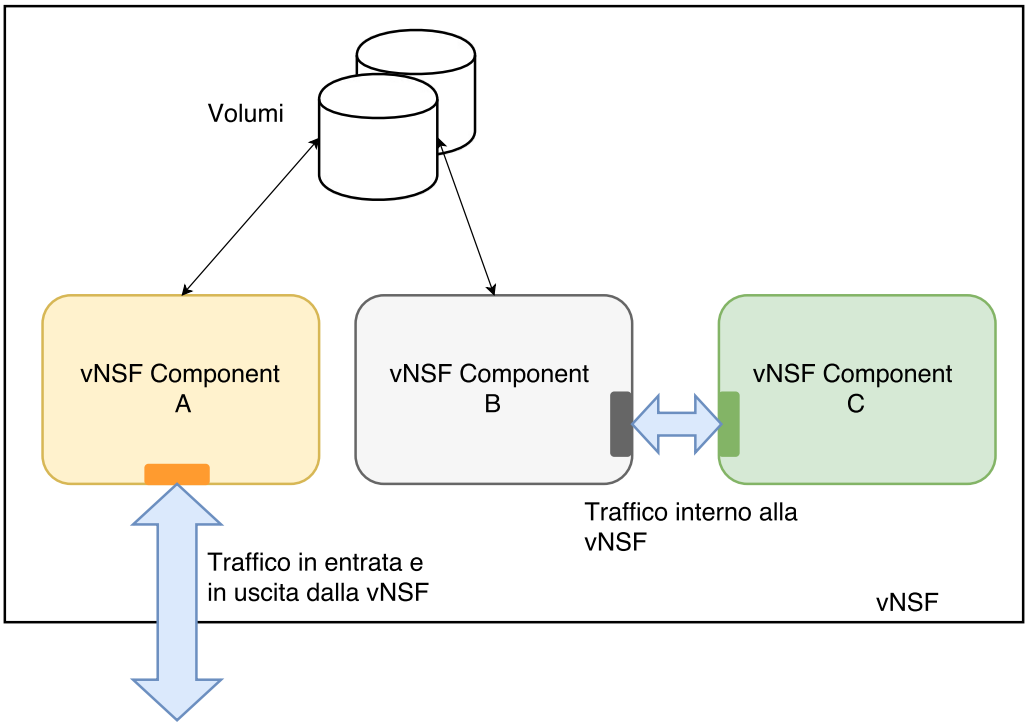

Figura 4.2. Architettura generica di una vNSF.

Si osserva la presenza di molteplici *vNSF Component* [105]. Essi suddividono le funzionalità di una vNSF in unità atomiche, utilizzando un approccio a microservizi (Capitolo 1). Pertanto, ogni vNSF Component si occupa dell'espletamento di un solo tipo di operazione es. log. I componenti possono essere eseguiti tutti all'interno dello stesso NFVI PoP o, mediante alcune estensioni discusse nella Sezione 4.5, su NFVI PoP diversi collocati in una LAN o in una WAN. Dal punto di vista implementativo, un vNSF Component è rappresentato da un singolo container e nel resto della trattazione i due termini saranno utilizzati in modo indifferente.

In secondo luogo, si nota come la comunicazione tra i vNSF Component può essere realizzata mediante interfacce di rete o tramite l'impiego di Volumi (Sezione 2.3.1).

Il primo caso implica l'installazione, in fase di avvio o a runtime, di opportune interfacce all'interno di container. Come riportato negli standard ETSI, tali interfacce sono trasparenti all'utilizzatore della vNSF e possono essere realizzate per la comunicazione all'interno dello stesso o diversi NFVI PoP [106, SWA-2 Interfaces]. Tale approccio favorisce, ad esempio, l'utilizzo di REST API per uno scambio flessibile di dati tra servizi in esecuzione nei vNSF Component. In entrambi i casi si rende imprescindibile la considerazione degli aspetti di sicurezza trattati nella Sezione 2.5.2. Difatti, è necessario limitare il rischio di ARP Poisoning revocando ai container l'utilizzo della Root Capability `CAP_NET_RAW` se non necessaria o connettendo allo stesso Virtual Bridge esclusivamente container appartenenti alla stessa vNSF.

L'utilizzo di volumi implica la presenza di un *binding*, ossia un collegamento, tra una directory all'interno del container e una al di fuori del suo Mount Namespace. In questo modo, è possibile condividere file tra vNSF Component senza dover intervenire sulla configurazione di rete interna. Si precisa che la directory *esterna* al container può essere sia locale, ossia salvata nel disco rigido dell'host, sia distribuita, mediante l'utilizzo di un apposito filesystem. In entrambi i casi è necessario l'utilizzo di meccanismi di DAC e MAC (Sezione 2.6.1) al fine di prevenire la scrittura e la lettura non autorizzata delle directory utilizzate da una vNSF.

Infine, l'architettura raffigurata presenta un punto di contatto tra un vNSF Component e l'esterno della vNSF. Esso permette l'esposizione della funzionalità della vNSF al di fuori del suo contesto. Analogamente a quanto riportato negli standard ETSI, tale interfaccia può essere utilizzata per la comunicazione della vNSF con un'altra vNSF, una *Physical Network Function* (PNF), ossia una funzione di rete non virtualizzata o un altro *endpoint* [106, SWA-1 Interfaces]. Uno dei modi più immediati per consentire il flusso di traffico proveniente dall'esterno è la predisposizione di un *port mapping*, ossia la configurazione del NFVI PoP in modo tale che traduca una porta in ascolto su un'interfaccia di rete fisica in all'interno di un Network Namespace. Dato che ogni porta esposta all'esterno provoca inevitabilmente un allargamento della superficie di attacco, è auspicabile che ogni vNSF sia progettata in maniera tale da ridurre al minimo i punti di accesso dall'esterno.

A differenza di quanto riportato dagli standard ETSI, non è definita, all'interno dell'architettura generica di una vNSF, un'interfaccia che consenta l'interazione con un'entità di management o di orchestrazione per il suo controllo e la sua riconfigurazione. Tale scelta deriva dalla necessità di utilizzare container leggeri, senza memoria degli stati precedenti e con una sola funzionalità [8, Best practices for writing Dockerfiles]. Pertanto, una vNSF così progettata può essere configurata esclusivamente in fase di avvio. Qualora vi fosse la necessità di modificarne la configurazione, sarà necessario procedere prima all'arresto e poi al riavvio della vNSF.

4.3.2 Definizione di security policy

Il processo di implementazione di policy di sicurezza (da qui *security policy*) all'interno di una vNSF richiede la presenza di un apposito linguaggio, già definito nel contesto del progetto *SECURity at the network EDge* (SECURED) [107]. Esso prende il nome di *Medium-level Security Policy Language* (MSPL).

Il linguaggio MSPL permette di esprimere security policy mediante file di configurazione indipendenti da qualsiasi implementazione software o hardware [108]. Tale disaccoppiamento si rivela particolarmente utile nel momento in cui si voglia provvedere ad una forma di *portabilità* della stessa policy tra implementazioni diverse. Si tratta di un linguaggio astratto costituito da una lista di asserzioni, ognuna delle quali referenzia un determinato tipo di *security control*. Pertanto, all'interno di un documento MSPL e con una sintassi generica, si esprimono security control quali:

- filtraggio di indirizzi IP;
- filtraggio di porte;
- controllo della presenza di un dato pattern nell'intestazione di una richiesta/risposta HTTP;
- controllo del tipo MIME nel corpo di una richiesta/risposta HTTP.

Nella Figura 4.3 è riportato un esempio di rappresentazione di security policy mediante il linguaggio MSPL.

Analizzandone i contenuti, si evince dal tag XML `<traffic-filter/>` che la security policy riguarda un'operazione di filtraggio. I tag `<ip/>` e `<block/>` lasciano intendere come il desiderio dell'utente sia vietare il traffico indirizzato o proveniente da determinati indirizzi IP, associati, ad esempio, a siti web illegali.

Sebbene pragmatiche, le direttive presenti in un file MSPL non possono essere utilizzate direttamente all'interno di una vNSF, poiché scritte in un formato indipendente da implementazioni.

```
<traffic-filter>
    <ip>
        <block>192.168.0.1</block>
        <block>192.168.0.2</block>
        <block>192.168.0.3</block>
    </ip>
</traffic-filter>
```

Figura 4.3. Esempio di security policy in linguaggio MSPL.

Difatti, non è plausibile la realizzazione di configurazioni immediatamente interpretabili da qualsiasi vNSF Component e vNSF. Pertanto, per poter attuare le security policy definite dall'utente, è necessario un ultimo processo di traduzione che converta le direttive MSPL in file di configurazione direttamente utilizzabili dai vNSF Component. È evidente come tale processo sia specifico per ognuna delle vNSF e necessiti della dovuta considerazione in fase di implementazione.

4.3.3 vNSF Reverse Proxy

Nella Sezione 4.3.1 è stata presentata la struttura *generica* di una vNSF, delineando il concetto di vNSF Component e le modalità con cui essi si interconnettono. Nella presente sezione si considera un caso specifico di vNSF, la vNSF Reverse Proxy. Quest'ultima contempla la presenza di tre vNSF Component:

Traduttore MSPL. Traduce configurazioni in MSPL in un formato comprensibile dal Reverse Proxy con WAF. Difatti, come riportato nella Sezione 4.3.2, tali configurazioni non possono essere utilizzate direttamente da un vNSF Component, poiché ideate per essere indipendenti da qualsiasi implementazione.

Reverse Proxy con WAF. Esegue le funzioni di proxy e filtraggio. Rappresenta il cuore della vNSF Reverse Proxy.

Collettore Log. Aggrega l'output del Reverse Proxy con WAF e lo traduce in un formato comprensibile da entità di management e auditing terze.

È opportuno segnalare che, data la presenza di due diverse modalità di interconnessione tra container, la vNSF Reverse Proxy può utilizzare, al suo interno, volumi, interfacce di rete o entrambi. È necessario comprendere come non sia possibile definire a priori la configurazione più efficiente, dal momento che è indispensabile considerare prima i dettagli implementativi di ogni singolo vNSF Component. Difatti, i meccanismi di I/O di un software possono essere predisposti per la lettura e scrittura di un file o di un socket.

Nel corso della presente sezione si farà riferimento ad un'architettura che prevede il solo utilizzo di volumi, essendo consapevoli che le considerazioni effettuate sono equivalenti a quelle di una configurazione con sole interfacce di rete o ibrida.

Nella Figura 4.4 sono rappresentate ordinatamente le interazioni che avvengono tra i suddetti vNSF Component nel processo di avvio di una vNSF.

Il punto 1 indica la scrittura di una configurazione in linguaggio MSPL all'interno del *MSPL Volume*. Tale operazione, preliminare all'avvio di qualsiasi vNSF Component, è compiuta da un eseguibile indipendente dalla vNSF in oggetto, il quale sarà trattato più approfonditamente nella Sezione 5.2.

Il punto 2 segnala l'*avvio* effettivo della vNSF Reverse Proxy, con la lettura, da parte del Traduttore MSPL, della configurazione salvata precedentemente nel MSPL Volume. Al termine

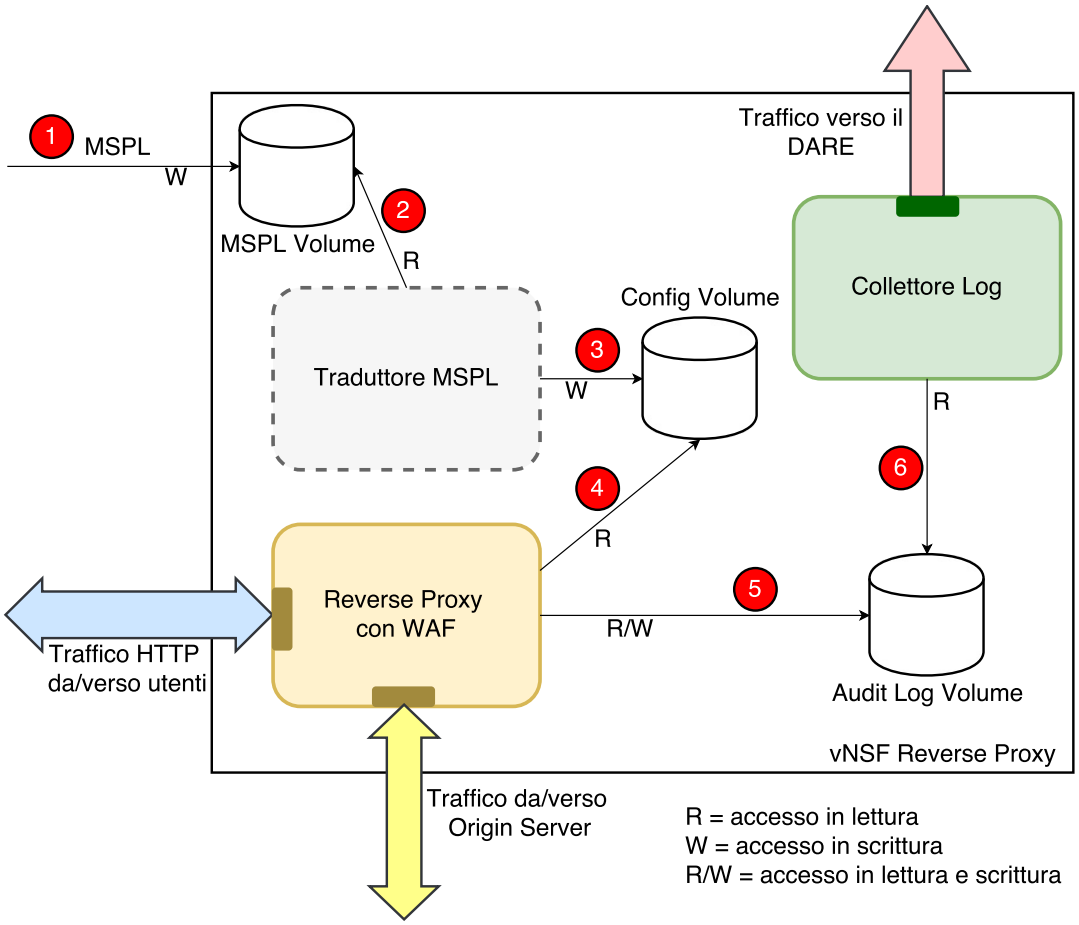

Figura 4.4. Architettura della vNSF Reverse Proxy.

della sua funzione, il Traduttore MSPL scrive l'output all'interno del *Config Volume* (punto 3) e si arresta. Pertanto, la vNSF a *runtime* non contempla la persistenza del Traduttore MSPL. Quest'ultimo aspetto è evidenziato dal fatto che il bordo della sua rappresentazione è tratteggiato.

Le considerazioni appena effettuate impongono una riflessione su un aspetto fondamentale della vNSF. L'arresto del Traduttore MSPL impedisce un cambiamento a runtime della security policy. Per rendere effettiva qualsiasi modifica al file MSPL è necessario riavviare la vNSF e invocare nuovamente il Traduttore MSPL. Non è possibile dunque, cambiare lo *stato* di una vNSF. Come anticipato nella Sezione 4.3.1, questo costituisce un punto di distacco dagli standard ETSI, i quali prevedono la presenza di un'interfaccia apposita all'interno della vNSF per la comunicazione con un'entità di management e orchestrazione [106, SWA-3 Interfaces].

Il punto 4 indica l'avvio del Reverse Proxy con WAF e la lettura della configurazione software scritta precedentemente nel Config Volume. Da questo momento, le funzioni richieste dalla vNSF sono attive. Dalla Figura 4.4 si nota come il Reverse Proxy con WAF presenti due flussi in entrata e due in uscita. Tali flussi rappresentano il traffico verso i client e la rete di backend. Sebbene graficamente agganciati a due interfacce di rete differenti, dal punto di vista implementativo è possibile che essi siano legati alla stessa interfaccia.

Nel punto 5 si vede come il Reverse Proxy con WAF abbia pieno accesso al *Audit Log Volume*. All'interno di esso sono custoditi i rilevamenti effettuati in base alla security policy fornita. Questi sono letti nel punto 6 dal terzo e ultimo vNSF Component avviato in questa vNSF: il Collettore Log.

Il Collettore Log aggrega il contenuto dell'Audit Log Volume e lo manipola affinché sia comprensibile da entità di management e auditing terze. Ad esempio, convoglia il contenuto dei rilevamenti

in documenti JSON e lo invia ad una specifica porta HTTP, in modo tale che si possano effettuare analisi successive.

In sintesi, si può affermare che la vNSF Reverse Proxy è costituita da tre vNSF Component avviati progressivamente, di cui il primo è presente esclusivamente nella fase di avvio. La condivisione dei dati avviene esclusivamente mediante l'impiego di volumi.

Per quanto concerne il networking, i punti di contatto con la rete Internet sono sul Reverse Proxy con WAF, per la gestione del traffico utente e sul Collettore Log, per l'invio dei dati all'esterno.

4.4 Architettura di deployment di vNSF basate su Docker

Nella Sezione 4.3 è stata presentata la struttura logica di una vNSF sia nel caso generale che nel caso specifico della vNSF Reverse Proxy.

Nella presente sezione si analizzano i file di configurazione che consentono di descrivere le specifiche tecniche del *deployment*, ossia l'avvio, di una vNSF. È importante premettere che, a differenza di quanto riportato nella Figura 4.2 e nella Figura 4.4, non esiste alcun blocco monolitico software che inglobi tutti i componenti di una vNSF. La presenza, nelle suddette figure, di un riquadro che contenga più moduli al suo interno è utilizzata esclusivamente per fini grafici. Inoltre, come anticipato nella Sezione 4.3.1, vi è una corrispondenza diretta tra un vNSF Component e un container. Pertanto, il deployment di una vNSF si concretizza nell'avvio di container con le relative strutture per la loro interconnessione.

Analogamente alla Sezione 2.3, si utilizzerà la piattaforma Docker come caso di studio, sebbene la sintassi qui presentata sia facilmente portabile ad alte piattaforme.

4.4.1 Componenti statiche e dinamiche del deployment

Al fine di facilitare l'analisi dei file di configurazione è opportuno introdurre i concetti di componente *statica* e componente *dinamica* del deployment.

Una componente statica è una singola direttiva o una porzione della configurazione definita dallo sviluppatore della vNSF e immutabile durante la fase di deployment. Essa descrive funzionalità di base della vNSF e una sua eventuale modifica ne può compromettere il corretto funzionamento. In questa categoria di direttive rientra la definizione dei vNSF Component di una vNSF.

Una componente dinamica è una singola direttiva o una porzione della configurazione definita dallo sviluppatore, ma modificabile per esigenze dettate dall'ambiente di esecuzione. Nel corso della trattazione si evince la presenza di diverse componenti di questo tipo, la cui modifica in fase di deployment è fortemente consigliata (Sezione 4.4.3). Un esempio è dato dall'impostazione di limiti alle risorse a disposizione di un container.

4.4.2 File manifest

Il file manifest rappresenta la configurazione principale della vNSF. Codificato in JSON, esso indica quali sono i vNSF Component che la costituiscono e le modalità con cui essi si interconnettono. Racchiude al suo interno componenti sia statiche sia dinamiche.

Nella Figura 4.5 è presentata la struttura di un file manifest. I primi campi sono utilizzati solo per l'identificazione della vNSF e della categoria a cui essa appartiene. Fatta eccezione per questi, si denota la presenza di tre diversi vettori: `Images`, `Containers` e `Volumes`.

Il vettore `Images` contiene una sequenza di Dockerfile da sottoporre all'operazione di Build (Sezione 2.3). Tale sezione è utilizzata da sviluppatori vNSF per poter costruire delle Immagini Docker non presenti su alcun Docker Registry poiché altamente specifiche per il contesto della vNSF di appartenenza. In aggiunta alla locazione del Dockerfile (campo `Path`), ogni indice del vettore `Images` raccoglie all'interno della struct `ImageBuildOptions` direttive per personalizzare l'operazione di

```
{
    "Vnsf":{
        "Name":"Sample-vNSF",
        "ID":"123456abcdef",
        "Type":"Reverse_Proxy_WAF",
        "Images":[
            {
                "Path":"...",
                "ImageBuildOptions":{...}
            },
             ...
        ],
        "Containers":[
            {
                "Name":"...",
                "Config":{...},
                "HostConfig":{...},
                "NetworkingConfig":{...},
                "Wait":...
            },
             ...
        ],
        "Volumes":[
            {
                "Name":"...",
                "Desc":"...",
                "Import":[...],
                "VolumesCreateBody":{...}
            },
            ...
        ]
    }
}
```

Figura 4.5. Struttura del file manifest.

Build. Per approfondire il contenuto di questa struttura dati si rimanda alla documentazione ufficiale [118]. La sezione `Images` rappresenta una componente statica della configurazione di una vNSF e, come tale, non sovrascrivibile.

Il vettore `Containers` elenca i vNSF Component, ossia i container, che costituiscono la vNSF. Ogni indice di questo vettore contiene diverse sezioni di configurazione. La sezione `Config` contiene informazioni sull'impostazione globale dei container, come l'Immagine di partenza. Nel caso di quest'ultima, si può referenziare un'Immagine presente in un Docker Registry o una elencata nella sezione `Images`. Tale porzione della configurazione è una componente statica. Le sezioni `HostConfig` e `NetworkSettings` contengono informazioni rispettivamente sulla configurazione del NFVI PoP e della rete in cui si trova il vNSF Component in oggetto. Al contrario della precedente, le impostazioni di queste strutture dati possono essere ampliate o sovrascritte in fase di deployment e costituiscono, pertanto, una componente dinamica. Anche in questo caso, è possibile approfondire il loro contenuto consultando la documentazione ufficiale [118].

Infine, il vettore `Volumes` determina i volumi da creare per consentire la comunicazione tra container e/o il salvataggio di dati in maniera persistente. È opportuno notare la presenza di una sezione `Import`, la quale offre la possibilità di copiare dei file esterni alla vNSF all'interno di un Volume. Tale voce è utilizzata, ad esempio, per la copia della configurazione MSPL della vNSF

```
vNSF_di_esempio.tar
├── manifest.json
├── Images_1/
│   ├── Dockerfile
│   └── Allegato_Dockerfile_1
├── Images_2/
│   ├── Dockerfile
│   ├── Allegato_Dockerfile_1
│   └── Allegato_Dockerfile_2
└── Images_3/
    └── Dockerfile
```

Figura 4.6. Impacchettamento di una vNSF all'interno di un archivio.

Reverse Proxy descritta nella Sezione 4.3.3. Ad eccezione della sezione `Import`, qualsiasi altra voce degli indici del vettore `Volumes` è una componente statica.

Salvataggio nel vNSF Store

Il file manifest costituisce una rappresentazione completa di una vNSF. Pertanto, una volta sviluppato può essere firmato digitalmente dal programmatore e salvato nel vNSF Store. In quest'ultimo, può essere indicizzato e selezionato da un utente.

È opportuno ricordare che il file manifest può contenere, nella sezione `Images`, una lista di Dockerfile da sottoporre all'operazione di Build. In tal caso, per poter essere utilizzati, è necessario che essi siano allegati al file manifest insieme alla lista di file da essi richiamati. Difatti, un Dockerfile può prevedere di copiare un determinato documento in una directory all'interno di un'Immagine. A tal proposito, una possibile soluzione è rappresentata dall'utilizzo di archivi. Nella Figura 4.6 è rappresento un esempio di rappresentazione vNSF con un archivio in formato *tar*.

In caso di rappresentazione di una vNSF mediante archivio, l'operazione di firma digitale sarà applicata all'intero archivio e non più al singolo file manifest.

4.4.3 File hostconfig

Nel contesto di deployment di una vNSF, il file hostconfig è adoperato per ampliare o sovrascrivere le componenti dinamiche del file manifest. È opportuno comprendere che l'esigenza di tale file nasce dall'impossibilità di definire, al momento dello sviluppo, alcune impostazioni dell'ambiente di deployment. Ad esempio, lo sviluppatore può definire una sottorete di default per la vNSF che sta sviluppando. In ambiente di produzione, si potrebbe voler inserire due vNSF all'interno della stessa sottorete e quindi modificare l'impostazione di default. Oppure, dato un container con un port mapping con la porta 80 del NFVI PoP, si vorrebbe modificare tale impostazione per poter riassegnare la mappatura con la porta 8080. Pertanto, è necessario che, in sede di deployment, il vNSFO o l'NFVI PoP siano in grado di ampliare o sovrascrivere le componenti dinamiche della configurazione.

Si consideri la struttura del file hostconfig riportata in Figura 4.7. Analogamente al file manifest, la codifica utilizzata per il file hostconfig è di tipo JSON. Al di sotto del nodo radice, si presenta un vettore di struct del tipo `ContainerOptions` e, all'interno di qualsiasi indice di questo, è possibile ritrovare due strutture omonime a quelle definite nel file manifest: `HostConfig` e `NetworkingConfig`. È di fondamentale importanza comprendere che ogni campo all'interno di queste struct nel file hostconfig sovrascriverà il corrispondente valore nel file manifest definito dallo sviluppatore della rispettiva vNSF. Pertanto, per evitare malfunzionamenti, è consigliabile apportare modifiche a tale file solo in piena consapevolezza delle modifiche che si stanno per apportare.

Definizione delle impostazioni di sicurezza

Tra i campi presenti all'interno della struct `HostConfig` del file hostconfig vi sono diverse impostazioni di sicurezza, la cui sovrascrittura in questo caso è fortemente consigliata, per evitare l'utilizzo di impostazioni di default non idonee allo specifico ambiente di deployment. Per approfondire la loro conoscenza è possibile consultare la documentazione ufficiale della Docker API [109]. A titolo d'esempio se ne riportano alcune:

`PidsLimit.` Definisce il numero massimo di PID utilizzabili all'interno di un container.

`Memory.` Determina, in byte, il limite massimo di memoria RAM utilizzabile da un container.

`CapDrop.` Rimuove Root Capability assegnate di default al container.

`SecurityOpt.` Consente di definire impostazioni per controllo degli accessi MAC, profili SECCOMP e User Namespace.

4.5 Estensione architettura di base e load balancing

La Sezione 4.3 e la Sezione 4.4 considerano la struttura logica di una vNSF e il suo deployment senza contestualizzarle in un determinato ambiente di esecuzione. Difatti, non è stata effettuata alcuna distinzione tra deployment di una vNSF in *locale*, ossia all'interno dello stesso NFVI PoP, o in un ambiente di calcolo distribuito es. cluster. Al fine di poter effettuare il deployment di una vNSF in quest'ultimo scenario, è indispensabile considerare aspetti quali l'ottimizzazione delle risorse, la *Fault-Tolerance* e la *scalabilità orizzontale*.

In un ambiente distribuito, l'ottimizzazione dell'hardware consiste in una suddivisione opportuna dei vNSF Component appartenenti alla stessa vNSF. Difatti, possono esservi componenti con

```
{
    "VnsfOptions":{
        "ContainerOptions":[
            {
                "Target":"reverse-proxy-waf",
                "HostConfig":{
                    "PortBindings":{...},
                    "SecurityOpt": [...],
                    "NetworkMode":"user",
                    "Memory":...,
                    "PidsLimit":...
                },
                "NetworkingConfig":{
                    "EndpointsConfig": {
                        "samplenet": {
                        "IPAMConfig":{...}
                    }
                }
            },
        ...
        ]
    }
}
```

Figura 4.7. Struttura del file hostconfig.

diverse esigenze in termini di risorse. A titolo d'esempio, si consideri il caso della vNSF Reverse Proxy. All'interno di questa, il Reverse Proxy con WAF necessita principalmente di potenza computazionale e ampia banda per gestire le transazioni HTTP. Il Collettore Log, al contrario, ha bisogno maggiormente di memoria RAM per l'aggregazione dei dati. Pertanto, è indubbio come una suddivisione mirata di tali componenti tra più host, con un diverso carico di lavoro o una diversa dotazione hardware, porti ad un ottimizzazione delle risorse.

La gestione della Fault-Tolerance consiste nell'assicurare il funzionamento di una vNSF, anche nel caso in cui vi siano dei malfunzionamenti all'interno del sistema host es. crash del sistema o danneggiamento di un disco locale. In quest'ultimo caso, ad esempio, è necessario che non si perdano file di log non ancora esaminati da entità di auditing e management terze.

La scalabilità orizzontale si concretizza nel considerare una vNSF come un insieme di una o più istanze identiche. Ogni istanza svolge in maniera completamente indipendente le funzioni di sicurezza che la vNSF si propone di offrire. Il numero di istanze deve poter essere diminuito o aumentato dinamicamente a seconda dell'esigenza momentanea. Nel caso della vNSF Reverse Proxy, è possibile replicarne più istanze in caso di un picco di richieste HTTP e porle nella rete di back-end di un Load Balancer (Sezione 4.5.3).

Per poter effettuare il deployment di una vNSF all'interno di un ambiente di produzione che tenga conto degli aspetti succitati, è necessario considerare altri componenti architetturali. Sebbene indipendenti dalla struttura logica di una vNSF, questi ultimi introducono aspetti che ne potenziano la funzionalità.

4.5.1 Utilizzo di Filesystem Distribuito

Un tipico caso di fault all'interno di un NFVI PoP è rappresentato dal malfunzionamento di un disco fisso. Gli scenari che si presentano in caso di tale eventualità sono molteplici, tra cui:

- Perdita file di log es. Audit Log;
- Perdita di file di input per una vNSF es. file da scansionare alla ricerca di malware;
- Perdita di Immagini Docker precedentemente scaricate da un Docker Registry remoto.

Le conseguenze derivanti dagli scenari appena elencati possono essere reversibili o irreversibili. La perdita di immagini Docker precedentemente salvate è reversibile, dal momento che è possibile procedere ad un nuovo download, al costo di un calo di prestazioni. Al contrario, la perdita di un file di un Audit Log non ancora sottoposto al DARE costituisce una conseguenza irrevesibile.

Al fine di scongiurare gli scenari succitati è necessario considerare all'interno dell'architettura sottostante la vNSF la presenza di un filesystem distribuito es. CephFS [110] e HDFS [111]. La loro configurazione esula dallo scopo di questa tesi.

L'utilizzo di un filesystem distribuito non comporta alcun cambiamento ai file manifest e host-config utilizzati per il deployment presentati nella Sezione 4.4. Nella Figura 4.8 è riportato un estratto di un file manifest. Si noti come sia sufficiente specificare il punto di Mount del filesystem distribuito nel binding del Volume per fare in modo che il Log del container `antivirus` non sia salvato nel disco fisso locale.

Considerando l'architettura della vNSF Reverse Proxy in Figura 4.4 si nota la presenza di tre diversi Volumi: MSPL Volume, Config Volume e Audit Log Volume. Mediante l'utilizzo di un filesystem distribuito si può garantire la condivisione dei suddetti tra i vari vNSF Component anche nel caso in cui i vari vNSF Component siano eseguiti su host differenti per scopi di ottimizzazione (Sezione 4.5). Pertanto, si nota come l'utilizzo di un filesystem distribuito in ambiente di produzione permetta allo stesso tempo di migliorare la Fault-Tolerance e lo sfruttamento delle risorse hardware a disposizione.

```
"Name":"antivirus",
"Config":{...},
"HostConfig":{
    "Binds":[
        "/path/to/localFS/input/:/opt/input/",
        "/path/to/cephFS/log/:/var/run/antivir/log/",
    ]
},
"NetworkSettings":{...}
```

Figura 4.8. Frammento di file manifest contenente binding di volumi.

4.5.2 Utilizzo di Overlay Network

Le impostazioni iniziali di Docker prevedono la possibilità di avviare container mediante i seguenti driver di rete:

`null`. Il container presenta al suo interno una sola interfaccia di loopback `lo`.

`bridge`. Il container è collegato al Virtual Bridge `docker0`.

`host`. Il container condivide lo stack di rete dell'host.

Tralasciando la configurazione `host`, ritenuta insicura nella Sezione 2.5.2, le rimanenti configurazioni sono idonee al solo networking all'interno dello stesso host. Tale configurazione potrebbe costituire un vincolo qualora si volesse ottimizzare la distribuzione dei vNSF Component sull'hardware a disposizione. Difatti, come specificato all'inizio della Sezione 4.5, è possibile che i componenti all'interno di una stessa vNSF abbiano diverse esigenze in termini di risorse.

Al fine di consentire il networking tra container in esecuzione su host diversi è possibile utilizzare il driver `overlay` [8, Docker container networking], il quale utilizza impiega il protocollo VXLAN.

Il protocollo VXLAN, descritto nella RFC-7348 [112], consente di incapsulare pacchetti del livello due dello stack di rete TCP/IP in pacchetti di livello tre. Nello specifico, consente la creazione di *tunnel* all'interno del payload di un pacchetto IP/UDP senza apportare la minima modifica alla preesistente architettura di routing. Il suo impiego è stato già indicato all'interno degli standard ETSI per fornire connettività tra VNF attraverso strati di virtualizzazione [113].

L'utilizzo di reti `overlay` in Docker, se non specificata la modalità *swarm* per l'orchestrazione di container, richiede la presenza di un opportuno *Key/Value Storage*, come etcd [114], Consul [115] o ZooKeeper [116]. Tali software consentono la memorizzazione di dati inerenti il *Control Plane* di questo tipo di rete in un formato chiave-valore. Pertanto, si estende l'architettura software vista finora con un altro componente non incluso nella piattaforma Docker.

Nella Figura 4.9 si osserva una possibile utilizzo di reti `overlay` nella vNSF Reverse Proxy. Tale scenario prevede una distribuzione dei vNSF Component descritti nella Sezione 4.3.3 in due NFVI PoP differenti, denominati A e B. Nello specifico, si nota la divisione, su sistemi differenti, del Reverse Proxy con WAF e del Collettore Log. In assenza di un filesystem distribuito (Sezione 4.5.1), è impossibile la trasmissione dell'Audit Log dal Reverse Proxy con WAF verso il Collettore Log. Pertanto, si procede ad un cambiamento architetturale che contempla, all'interno della vNSF Reverse Proxy, l'utilizzo di interfacce di rete non previste nella Figura 4.4. In altre parole, si migra dalla condivisione dell'Audit Log mediante file salvati sull'Audit Log Volume in un flusso di dati mediante una rete `overlay`.

Con riferimento alla Figura 4.9, una possibile configurazione dei vNSF Component succitati è la seguente:

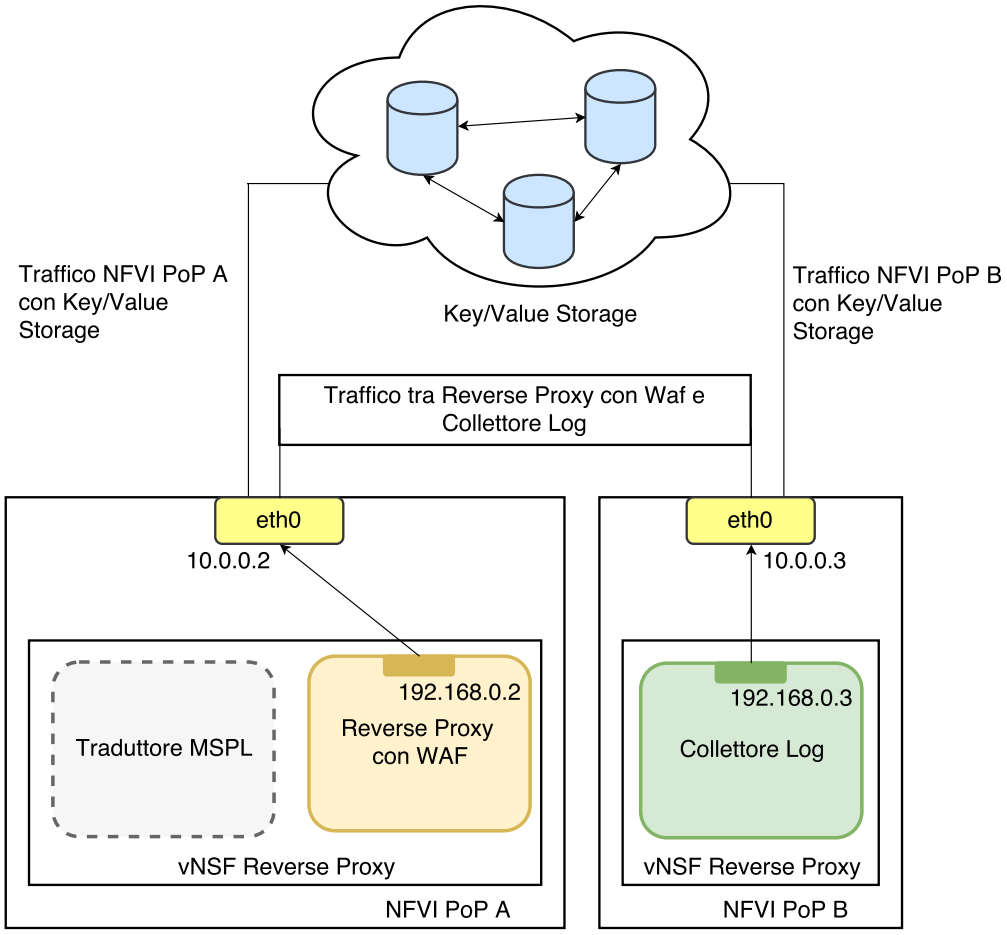

Figura 4.9. vNSF Reverse Proxy suddivisa in due parti comunicanti con overlay network.

- Reverse Proxy con WAF:

  Indirizzo MAC : c1:c1:c1:c1:c1:c1;

  Indirizzo IP: 192.168.0.2;

- Collettore Log:

  Indirizzo MAC : c2:c2:c2:c2:c2:c2;

  Indirizzo IP: 192.168.0.3;

  Porta ricezione Audit Log: 7890.

Utilizzando tale configurazione, un frammento dell'Audit Log è trasmesso in pacchetti aventi la struttura riportata in Figura 4.10.

4.5.3 Load Balancer

Nell'introduzione al presente Capitolo è stata trattato il concetto di scalabilità orizzontale. Esso prevede la possibilità di rimuovere o aggiungere istanze della *stessa* vNSF a seconda delle esigenze, assicurando che non vi sia uno spreco di risorse in caso di basso carico o una carenza in caso di necessità. Pertanto, è evidente come l'architettura *interna* di una vNSF sia indipendente da questo aspetto.

Nel caso della vNSF Reverse Proxy, la scalabilità orizzontale si concretizza nella possibilità di avere un numero variabile di istanze a seconda del volume di traffico in entrata e in uscita dal

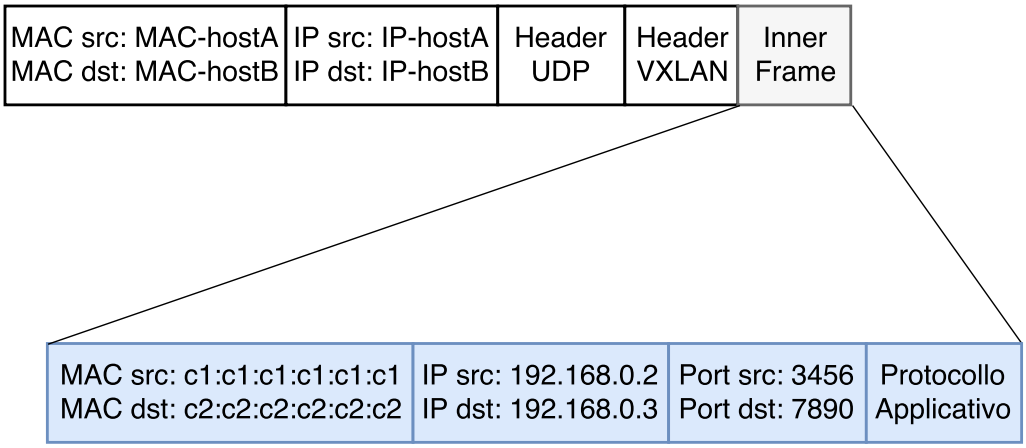

Figura 4.10. Pacchetto diretto dal NFVI PoP A al NFVI PoP B che incapsula dati dell'Audit Log.

Reverse Proxy con WAF. Difatti, è necessario considerare come il numero di transazioni in un dato momento sia determinato da diversi fattori es. fascia oraria. Pertanto, è necessario che la vNSF Reverse Proxy sia reattiva a tali cambiamenti, precludendo all'utente la possibilità di percepire una qualità del servizio differente in momenti differenti. Al fine di poter soddisfare il requisito di scalabilità orizzontale è necessario introdurre un componente esterno: il Load Balancer.

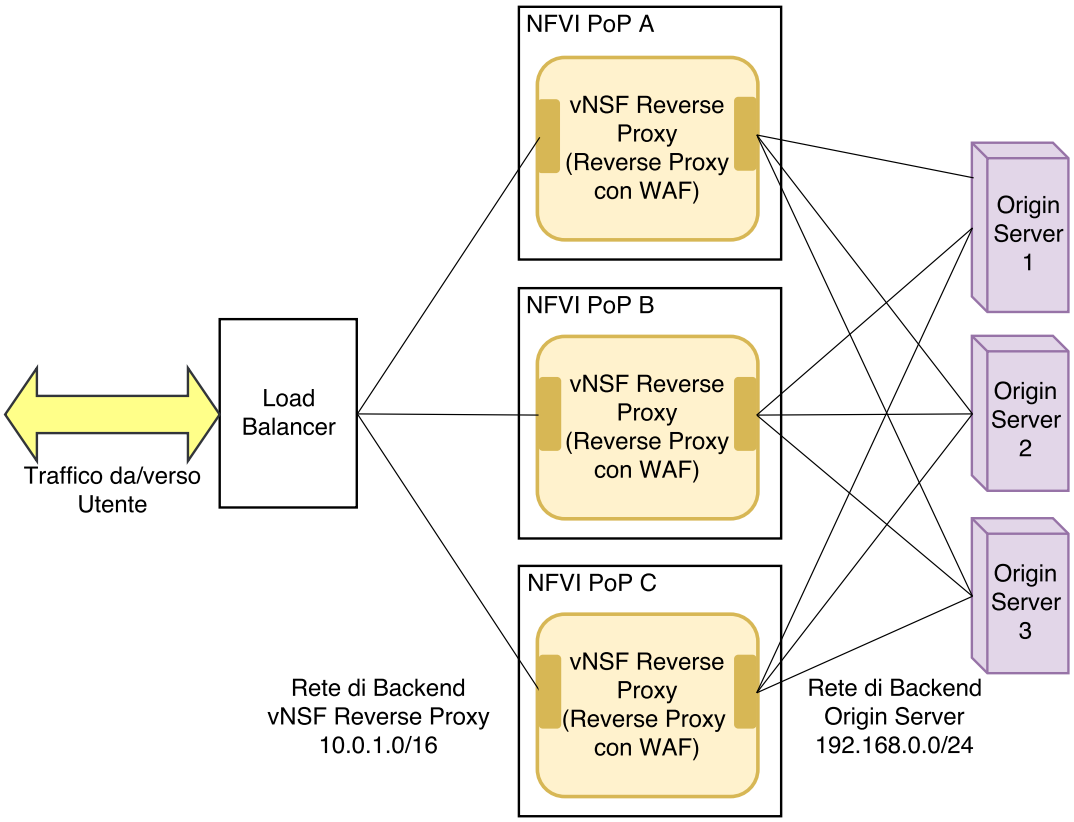

Figura 4.11. Esempio di configurazione vNSF Reverse Proxy con Load Balancer.

Nella Figura 4.11 è raffigurato uno scenario con tre istanze della vNSF Reverse Proxy raggiungibili da un Load Balancer. Per semplificare la rappresentazione la vNSF Reverse Proxy è stata rappresentata con il solo Reverse Proxy con WAF, dando per assunta la presenza del Traduttore MSPL e del Collettore Log. A differenza dell'architettura riportata in Figura 4.4 si nota come il

Reverse Proxy con WAF non disponga più di un contatto diretto con il traffico utente. Difatti, è stata creata una seconda rete privata, anche questa considerabile come backend, costituita da tutte le istanze della vNSF Reverse Proxy. Queste ultime sono utilizzabili esclusivamente attraverso il Load Balancer, che diviene ora il nuovo punto di ingresso del traffico utente.

L'implementazione del Load Balancer è indipendente dalla struttura rappresentata in Figura 4.11. Esso può essere uno dei software Load Balancer elencati nella Sezione 3.2 in esecuzione su un altro host o un ulteriore container. In quest'ultimo caso la rete di backend tra istanze potrebbe essere realizzata mediante il driver `overlay` discusso nella Sezione 4.5.2.

In conclusione, è bene enfatizzare che l'utilizzo di più istanze di una stessa generica vNSF consente anche di migliorare la Fault-Tolerance della soluzione proposta. Difatti, la presenza di varie istanze permette di assicurare la continuità del servizio in caso di crash dell'host o di una singola istanza. Pertanto, è evidente come sia necessario distribuire varie istanze della stessa vNSF su NFVI PoP diversi, al fine di limitare la probabilità che un guasto di qualsiasi natura causi un disservizio.

Capitolo 5

Implementazione

5.1 Overview

Nella Sezione 4.4 è stata preannunciata l'impossibilità di identificare la vNSF come un blocco monolitico, sottolineando come questa sia costituita da un insieme di diversi componenti collegati tra loro. Inoltre, è stata assodata la corrispondenza biunivoca tra vNSF Component e container. Successivamente, sono state analizzate diverse modalità di interconnessione tra questi e valutate possibili estensioni.

Il presente Capitolo si pone l'obiettivo di presentare un'implementazione della vNSF Reverse Proxy in qualità di *Proof of Concept* (PoC). Questa segue ovviamente i principi architetturali già delineati nel corso del Capitolo 4. Prima di procedere all'analisi dettagliata di ogni vNSF Component all'interno di questo caso di studio, si presenta un software ausiliario per l'avvio di vNSF, opportunamente progettato e implementato nel corso del presente lavoro di tesi: il vNSF Controller.

5.2 vNSF Controller

Nella Figura 4.1 è stata esemplificata l'interazione tra alcuni degli aspetti principali di un modello Secaas basato su un'infrastruttura NFV. Tale rappresentazione maschera diversi componenti software che intervengono sia in fase di avvio di una vNSF sia nella sua gestione durante l'esecuzione. Alcuni di essi sono costantemente attivi come *daemon* all'interno della Network Infrastructure, altri sono richiamati esclusivamente in risposta a particolari eventi. Tra gli obiettivi del presente lavoro di tesi rientra lo sviluppo e l'analisi di uno rientrante nella seconda categoria: il vNSF Controller.

Il vNSF Controller è un componente software utilizzabile all'interno di un NFVI PoP. Quest'ultimo equipaggia tutti i software necessari per garantire il corretto funzionamento di una vNSF, a partire dall'installazione della piattaforma Docker.

Il vNSF Controller si propone come interfaccia tra un orchestratore e *qualsiasi* tipo di vNSF, occupandosi materialmente dell'espletamento delle seguenti funzioni:

- avvio di una vNSF;
- lettura dello stato di una vNSF;
- arresto di una vNSF.

Il vNSF Controller è stato implementato nel linguaggio di programmazione open-source *Go* [117]. Go presenta una sintassi simile a quella utilizzata nel linguaggio C, è compilato staticamente ed incorpora in file eseguibili tutto il necessario per l'esecuzione, ad eccezione di eventuali librerie

dinamiche. Tra le funzionalità di Go rientra anche il Garbage Collector, ossia un sistema che libera porzioni di memoria non più utilizzate da un'applicazione in esecuzione.

Docker fornisce in Go un *Software Development Kit* (SDK), ossia un set di strumenti per la programmazione, utilizzabile per lo sviluppo di applicazioni che utilizzino container. Si segnala la presenza di SDK *non ufficiali* scritte in altri linguaggi di programmazione es. C/C++. La loro funzionalità, tuttavia, non è stata testata dal team di sviluppo di Docker [8, Develop with Docker Engine SDKs and API].

L'avvio e il controllo dei singoli vNSF Component, e dunque della vNSF stessa, è effettuato dal vNSF Controller interagendo con il Docker Daemon (Sezione 2.3) in esecuzione sul NFVI PoP. Precisamente, il vNSF Controller invoca metodi appartenenti al *Docker Engine Go SDK* [118]. Questi ultimi utilizzano la Docker API (Sezione 2.3) per interfacciarsi con il Docker Daemon.

Come già anticipato, il vNSF Controller non è stato concepito per essere costantemente in esecuzione come daemon all'interno del NFVI PoP. Questo preclude la possibilità che sia contattato come servizio in ascolto su un socket TCP/UDP o UNIX. Essendo attivo soltanto durante l'espletamento delle funzioni succitate, esso è utilizzabile lanciando il suo file eseguibile con una qualsiasi funzione Linux appartenente alla famiglia `exec()`.

Il vNSF Controller, una volta avviato, svolge le operazioni richieste in maniera sincrona e termina immediatamente l'esecuzione. Ogni singola funzione tra quelle prima elencate richiede una diversa istanza di tale software. Ad esempio, l'avvio di due vNSF distinte richiede due esecuzioni del vNSF Controller. Allo stesso modo, l'avvio di una vNSF e la lettura dello stato della stessa devono essere effettuate con due processi utente distinti.

Data la sua natura effimera e mirata, il vNSF Controller presenta una struttura semplice. Esso utilizza un unico thread dal momento che le funzioni che svolge necessitano il rispetto di un preciso ordine e non possono essere eseguite parallelamente.

Le versioni con cui è stato effettuato lo sviluppo e il test di vNSF Controller sono le seguenti:

- Go versione 1.8.1 per linux/amd64;
- Docker API versione 1.32;
- Docker 17.09.0-ce.

5.2.1 Avvio di una vNSF

L'avvio di una vNSF rappresenta la funzione più complessa e delicata che il vNSF Controller si propone di svolgere. La sua esecuzione restituisce una vNSF funzionante in tutti i suoi aspetti e non più modificabile nella configurazione. Il motivo di tale restrizione è da ricercare nelle già citate raccomandazioni (Sezione 4.3.3) di utilizzare container senza stati, effimeri e con un singolo scopo, riportate nella documentazione ufficiale di Docker [8, Best practices for writing Dockerfiles].

La funzione di avvio di una vNSF riceve almeno tre diversi input: la rappresentazione della vNSF mediante singolo file manifest o archivio (Sezione 4.4.2), il file hostconfig e la configurazione in linguaggio MSPL (Sezione 4.3.2). La richiesta e la ricezione di tali file sono svolte in un momento antecedente l'avvio del vNSF Controller. È l'NFVI PoP a prelevare la rappresentazione della vNSF dal vNSF Store e la configurazione MSPL, unita al file hostconfig, dal vNSFO.

Nello stato attuale, il vNSF Controller *non* effettua alcun controllo sulla validità della firma digitale associata alla vNSF e all'integrità dei file ricevuti dal vNSFO. Pertanto, si da per assunto che tali controlli siano effettuati prima di invocare il vNSF Controller da parte del NFVI PoP o di qualsiasi altro componente opportunamente configurato.

Una volta completato lo startup, il vNSF Controller inizia l'operazione di avvio di una vNSF invocando la funzione `Vnsf_launcher(...)`. Tale metodo si occupa di espletare sequenzialmente le seguenti operazioni:

- elaborazione del file manifest;

- elaborazione del file hostconfig;
- avvio dei componenti.

Elaborazione del file manifest

Il file manifest, descritto nella Sezione 4.4.2, indica quali sono i vNSF Component di una vNSF e in che modalità si interconnettono. Tramite la funzione `Parse_vnsfconf(...)`, il contenuto di tale file, in formato JSON, è decodificato e le sue sezioni memorizzate all'interno di opportune strutture dati. Pertanto, al termine di tale processo, le sezioni `Images`, `Containers` e `Volumes` del file manifest sono presenti in memoria RAM sotto forma di mappe.

È bene precisare che durante questa fase si crea anche una quarta struttura dati, costituita dalle Immagini referenziate nella sezione `Containers` e non presenti nella sezione `Images`. Se non specificato un Docker Registry, tali Immagini sono ricercate dal vNSF Controller all'interno del Docker Hub.

Elaborazione del file hostconfig

Il file hostconfig, descritto nella Sezione 4.4.3, è utilizzato per sovrascrivere le componenti dinamiche della configurazione. L'elaborazione di tale file, ad opera della funzione `Parse_dockerhost(...)`, segue immediatamente quella del file manifest.

In primo luogo, si provvede alla decodifica, dal formato JSON, del vettore `ContainerOptions`. Ogni elemento al suo interno ha una corrispondenza biunivoca con un altro presente all'interno del vettore `Containers` del file manifest.

Successivamente, si procede ad un aggiornamento delle mappe create durante l'elaborazione del file manifest con i dati appena decodificati. Data la delicatezza di questa operazione, si sottolinea nuovamente l'importanza di scegliere con cura le componenti dinamiche della configurazione da modificare con il file hostconfig. Inoltre, si ricorda come il vNSF Controller non è attualmente in grado di verificare l'integrità di tale file. Pertanto, si da per assunto che tale controllo sia fatto in un momento precedente da un qualsiasi altro software.

Avvio dei componenti

A differenza delle fasi precedenti, l'avvio dei componenti non necessita di ulteriori input. Essa è eseguita all'interno della già citata `Vnsf_launcher(...)`. A partire dalle strutture dati precedentemente create, l'avvio dei componenti rispetta il seguente ordine:

1. build dei Dockerfile presenti;
2. download delle immagini dal Docker Hub o da un altro Docker Registry indicato;
3. creazione dei volumi;
4. avvio dei container.

Durante la fase della loro creazione, è possibile copiare file all'interno dei volumi. Questo si rivela utile nel momento in cui vi siano container che necessitano di determinati input già al loro avvio. Ad esempio, la presenza del file MSPL nel MSPL Volume della vNSF Reverse Proxy.

In ognuna delle fasi succitate, il vNSF Controller dialoga con il Docker Daemon mediante funzioni appartenenti al già citato Docker Engine Go SDK.

La fase di avvio dei componenti termina con la scrittura del file *vNSF_ID* `_status.json`. Esso si limita a riportare, in un formato JSON, gli identificativi dei container e dei Volumi utilizzati all'interno della vNSF. Ultimata tale operazione, la vNSF può essere utilizzata e il vNSF Controller si arresta.

5.2.2 Lettura dello stato e arresto di una vNSF

Lo stato di una vNSF può essere recuperato specificando il parametro `--status=true` e il file *vNSF_ID* `_status.json` all'avvio del vNSF Controller. Lo stato di una vNSF è rilevato all'interno della funzione `Check_Vnsf_Status(...)` e restituisce esclusivamente il valore di `HEALTHCHECK` per i container che ne siano provvisti.

`HEALTHCHECK` è una direttiva utilizzabile all'interno di un Dockerfile. Essa consente di specificare un comando da eseguire all'interno di un container per verificare che questo sia ancora attivo. In questo modo, è possibile rilevare eventuali situazioni di errore all'interno di un'entità [8, Dockerfile reference].

Nell'implementazione attuale, lo stato di una vNSF può essere interrogato periodicamente per controllare l'attività dei vNSF Component. Si tratta di una procedura passiva, dal momento che il vNSF Controller si limita a notificarne lo stato al NFVI PoP o al vNSFO. È opportuno che eventuali provvedimenti, ad esempio il riavvio della vNSF in stallo, siano ordinati dal vNSFO.

L'arresto di una vNSF in esecuzione è ordinato specificando il parametro `--stop=true` e il file *vNSF_ID* `_status.json`. La funzione `Stop_Vnsf(...)` provvede all'arresto immediato dei container, ma non alla cancellazione dei Volumi. Per tale funzionalità, è necessario specificare la direttiva `--deletevolume=true`. In questo modo, è possibile conservare dati appartenenti all'elaborazione svolta dai vNSF Component per eventuali investigazioni successive.

5.3 vNSF Reverse Proxy: Panoramica

Nella Sezione 4.3.3 è stata presentata l'architettura della vNSF Reverse Proxy. Essa è costituita da tre vNSF Component: il Traduttore MSPL, il Reverse Proxy con WAF e il Collettore Log. Ognuno di essi è analizzato nel dettaglio nelle Sezioni seguenti. Nella presente sezione, invece, si considerano la rappresentazione della vNSF Reverse Proxy all'interno del vNSF Store e una possibile configurazione del file hostconfig, tale da rispettare le best-practice di sicurezza trattate nel corso del Capitolo 2.

5.3.1 File manifest della vNSF Reverse Proxy

Nella Sezione 4.4.2 è stata presentata la struttura generica di un file manifest, indicando il ruolo di ciascuna delle porzioni che lo costituiscono. Nella Figura 5.1 è riportata una versione semplificata del manifest della vNSF Reverse Proxy. Per una versione più completa, si può far riferimento all'Appendice B.7.8.

Il vettore `Images`, definito a partire dalla riga 4, contiene tre elementi. Dai loro identificativi, si comprende come ognuno di essi appartenga ad uno specifico vNSF Component. La presenza di tutti i componenti della vNSF Reverse Proxy all'interno del vettore `Images` non è da intendersi come requisito. Difatti, come già affermato nella Sezione 4.4.2, tale segmento del file JSON serve ad indicare Dockerfile non ancora sottoposti all'operazione di Build, poiché non presenti in alcun Docker Registry. Nel caso della vNSF Reverse Proxy, nello stato attuale, nessuna delle tre Immagini di partenza dei container è salvata presso un Docker Registry. Pertanto, è necessario che si segnali al vNSF Controller la necessità di effettuare tre operazioni di Build, una per ogni vNSF Component.

All'interno di ciascun indice del vettore `Images`, è possibile notare la presenza della direttiva `Path`. Essa è utilizzata per indicare il percorso del Dockerfile da sottoporre all'operazione di build. Nello specifico, essa determina l'archivio in formato *tar* contenente il Dockerfile ed eventuali allegati.

Il vettore `Containers` contiene un elemento per ciascuno dei tre vNSF Component della vNSF Reverse Proxy. Le direttive `Image`, presenti rispettivamente alle righe 20, 30 e 37, enfatizzano tale corrispondenza. Si noti che nella Sezione `Binds` di ciascun indice del vettore `Containers` è specificato il collegamento tra il vNSF Component e i relativi volumi. È evidente come la condivisione dei volumi tra container segua lo schema proposto in Figura 4.4.

```
{
    "Vnsf":{
        "Id":"E0BB2F1E8C7A561E9284A141AEDBBE2C",
        "Images":[ {
            "Path":"./MSPLTranslatorDocker/MSPLTranslatorDocker.tar",
            "ImageBuildOptions":{
                "Tags":["mspltranslator_image"] }
        }, {
            "Path":"./CollectorDocker/CollectorDocker.tar",
            "ImageBuildOptions":{
                "Tags":["collector_image"] }
        }, {
            "Path":"./ReverseProxyDocker/ReverseProxyDocker.tar",
            "ImageBuildOptions":{
                "Tags":["reverseproxy_image"] }
        } ],
        "Containers":[ {
            "Name":"mspl-translator",
            "Config":{
                "Image":"mspltranslator_image",
                "Cmd":["./mspltranslator","-input",
                    "/opt/mspl-volume/{{.Mspl}}"] },
            "HostConfig":{
                "Binds":[ "MSPL_Volume:/opt/mspl-volume",
                    "Config_Volume:/opt/proxy-conf:Z" ] },
            "Wait":true
        }, {
            "Name":"reverse-proxy-waf",
            "Config":{
                "Image":"reverseproxy_image" },
            "HostConfig":{
                "Binds":[ "Config_Volume:/opt/proxy-conf",
                    "Audit_Log_Volume:/var/log/httpd:Z" ] }
        }, {
            "Name":"collector",
            "Config":{
                "Image":"collector_image" },
            "HostConfig":{
                "Binds":[ "Audit_Log_Volume:/opt/input/",
                    "Output_Log_Volume:/opt/output:Z" ] }
        } ],
        "Volumes":[ {
            "Name":"MSPL_Volume",
            "Import":["{{.Mspl}}"] },
        {
            "Name":"Config_Volume"},
        {
            "Name":"Audit_Log_Volume"},
        {
            "Name":"Output_Log_Volume"} ]
    }
}
```

Figura 5.1. Versione semplificata del manifest della vNSF Reverse Proxy.

Nelle righe 25, 33 e 40 della Figura 5.1 si nota la presenza dei caratteri `:Z` in coda alle direttive di binding dei Volumi `MSPL_Volume`, `Config_Volume` e `Audit_Log_Volume`. Tale sequenza serve a definirela label del MAC (Sezione 2.6.1). Il Docker Daemon, giacché in possesso dei privilegi di amministratore, assegna al Volume una label predefinita o una casuale qualora non vi sia una direttiva `SecurityOpt` (vedi Sezione 5.3.2). In questo modo, si assicura che l'accesso al Volume sia protetto mediante meccanismi di Type Enforcement e di Multi Category Support. Visto in altri termini, la sequenza `:Z` indica che il Volume a cui essa è applicata è etichettato con `svirt_sandbox_file_t:s0:`*`categoria`*. Questo garantisce sia protezione da processi in esecuzione sull'host, con l'indicazione di tipo `svirt_sandbox_file_t`, sia da altri container, mediante l'utilizzo della *`categoria`*. Si rimanda alla Sezione 5.3.2 per un esempio.

Alla riga 26, è presente la direttiva `Wait:True`. Essa segnala al vNSF Controller la necessità di attendere la terminazione del container prima di procedere all'avvio del successivo nel vettore `Containers`. Difatti, come riportato in Sezione 4.3.3, il Traduttore MSPL (qui `mspl-translator`) rimane attivo esclusivamente nelle fasi di avvio e la sua terminazione (senza errori) segnala la disponibilità di una configurazione immediatamente utilizzabile dal Reverse Proxy con WAF. Pertanto, la direttiva `Wait:True` evita che si proceda all'avvio del `reverse-proxy-waf` senza che vi sia una configurazione valida nel MSPL Volume. In caso di errore, il vNSF Controller interrompe l'avvio della vNSF e ritorna un codice di errore.

La Sezione `Volumes` all'interno del manifest della vNSF Reverse Proxy si limita ad evidenziare la presenza di quattro diversi volumi. In fase implementativa è stato aggiunto l'`Output_Log_Volume`. Esso contiene i risultati dell'azione del Collettore Log, in attesa che essi siano letti o inviati al DARE. La sua funzione non è indispensabile al funzionamento della vNSF Reverse Proxy, dal momento che si tratta esclusivamente di una copia locale di dati già elaborati.

Da notare la direttiva `Import` alla riga 44: essa permette di specificare, mediante un segnaposto detto *placeholder*, il nome del file contenente la configurazione MSPL della vNSF Reverse Proxy. Nel rispetto di quanto riportato in Figura 4.4, è cura del vNSF Controller la scrittura di tale configurazione all'interno del MSPL Volume, in modo tale che possa essere letta dal Traduttore MSPL al suo avvio.

Rappresentazione nel vNSF Store

Dal momento che il file manifest presenta diversi riferimenti a Dockerfile nella Sezione `Images`, risulta evidente come la vNSF Reverse Proxy debba essere rappresentata all'interno del vNSF Store come un archivio contenente più file. Nella Figura 5.2 è rappresentata la struttura di tale archivio. I file fin qui non nominati verranno esaminati nelle sezioni successive.

Inoltre, il contenuto dell'archivio rappresenta la parte più *statica* della vNSF Reverse Proxy. Essa può essere sottoposta a firma digitale per fare in modo che ne siano garantite autenticità, integrità e paternità dello sviluppatore. Come già ampiamente trattato, la parte *dinamica* della configurazione è rappresentata mediante un secondo documento, ossia il file hostconfig.

5.3.2 Esempio di hostconfig

Nella Sezione 4.4.3 è stata presentata la struttura del file hostconfig per la sovrascrittura delle componenti dinamiche del file manifest. Nella Figura 5.3 è riportato un esempio di possibile file hostconfig da allegare alla rappresentazione della vNSF Reverse Proxy appena discussa. È opportuno premettere come esso costituisca una linea guida per la configurazione e possa essere modificato per esigenze dettate dall'ambiente di produzione.

All'interno del vettore `ContainerOptions` si contano due elementi, la cui presenza è enfatizzata dalla direttiva `Target`. Quest'ultima permette di identificare il container le cui componenti dinamiche della configurazione saranno aggiornate.

Ogni elemento di `ContainerOptions` contiene una struct `HostConfig`, della quali si invita nuovamente ad approfondire la conoscenza consultando la documentazione ufficiale della Docker API [109]. Si riscontra la mancanza della struttura `NetworkingConfig`, presente invece in Figura 4.7.

```
vNSF Reverse Proxy.tar.gz
├── manifest.json
├── MSPLTranslatorDocker/
│   └── MSPLTranslatorDocker.tar
│       ├── Dockerfile
│       └── mspltranslator
├── ReverseProxyDocker/
│   └── ReverseProxyDocker.tar
│       └── Dockerfile
└── CollectorDocker/
    └── CollectorDocker.tar
        ├── Dockerfile
        ├── input_conf
        ├── output_conf
        └── template_virtualhost
```

Figura 5.2. Rappresentazione vNSF Reverse Proxy nel vNSF Store.

```
1  {
2      "VnsfOptions":{
3          "ContainerOptions":[ {
4              "Target":"reverse-proxy-waf",
5              "HostConfig":{
6                  "PortBindings":{
7                      "80/tcp":[ {
8                          "HostPort":"11082" } ] },
9                  "SecurityOpt": [ "label:level:s0:c676" ],
10                 "Memory":1073741824,
11                 "PidsLimit":1000,
12                 "CapDrop":["NET_RAW"] }
13         }, {
14             "Target":"collector",
15             "HostConfig":{
16                 "SecurityOpt": [ "label:level:s0:c676" ] ,
17                 "Memory":1073741824,
18                 "PidsLimit":1000,
19                 "CapDrop":["NET_RAW"] }
20         } ]
21     }
22 }
```

Figura 5.3. Esempio di file hostconfig per la vNSF Reverse Proxy.

Questa è giustificata dall'assenza di connessioni mediante interfacce di rete nella vNSF Reverse Proxy rappresentata in Figura 4.4.

Il vNSF Component denominato `reverse-proxy-waf`, presenta una configurazione di *port binding*, visibile a partire dalla riga 6. Tale impostazione consente di impiantare una corrispondenza tra una porta di rete del NFVI PoP e una porta all'interno del container. Nel caso in esame, si procede a creare un collegamento tra la porta `11082` del NFVI PoP con la porta `80` del container.

Le righe 9 e 16 specificano, nel contesto della direttiva `SecurityOpt`, la label SELinux da applicare ai processi in esecuzione nel `Target`, ossia nel relativo container. Si consideri la stringa

riportata alla riga 9: `"SecurityOpt": ["label:level:s0:c676"]`. Implicitamente, essa indica di assegnare ai processi all'interno del container il tipo `varname_lxc_net_t`, un grado di sensibilità pari a `s0` e una categoria di esempio `c676`. Questo comporta che il container denominato `reverse-proxy-waf` può accedere a tutti i file di tipo `svirt_sandbox_file_t` in caso di solo Type Enforcement e file dello stesso tipo, ma con sensibilità `s0` e categoria `c676` in caso di Multi Category Support.

La direttiva `SecurityOpt`, unita alla sequenza di caratteri `:Z` presente in Figura 5.3, assicura che i Volumi abbiano lo stesso tipo, la stessa sensibilità e categoria specificati per il container. Considerando l'esempio di hostconfig in Figura 5.3, è evidente come i volumi Config Volume e Audit Log Volume della vNSF Reverse Proxy siano etichettati allo stesso modo, permettendo al `reverse-proxy-waf` e al `collector` di condividerli e renderli inaccessibili ad altri container eventualmente in esecuzione sullo stesso host.

L'assegnazione di label per MAC è una componente fortemente dinamica della configurazione della vNSF. Non è conveniente consentire un'assegnazione statica di tali direttive di sicurezza da parte del programmatore della vNSF. Pertanto, è opportuno che tale aspetto sia curato da un'entità che abbia una maggiore conoscenza del contesto di sicurezza in cui sono eseguite le vNSF, come il vNSFO o l'NFVI PoP.

Le righe 12 e 19 indicano al Docker Daemon di negare ai due container la Root Capability `CAP_NET_RAW`. Nella Sezione 2.5.2 è stato affermato che tale Root Capability consente l'accesso a socket di tipo sia *PACKET* sia *RAW*, ossia socket che inoltrino pacchetti dotati o meno della formattazione data da un livello di trasporto es. TCP. Date le sue potenzialità, essa può essere sfruttata per attacchi di ARP Poisoning. Nel caso della vNSF Reverse Proxy, nessuno dei componenti necessita di utilizzare socket di tipo RAW. Pertanto, è opportuno ridurre la superficie di attacco dei vNSF Component negando tale Root Capability. È opportuno precisare che il divieto di utilizzare socket di tipo RAW impedisce il funzionamento nel container di strumenti di diagnosi come `ping`. Pertanto, si enfatizza come la configurazione del file hostconfig dipenda dalle esigenze dell'ambiente di produzione e possa essere modificata in diversi aspetti.

All'interno della struct `HostConfig` si ritrovano le direttive `Memory` e `PidsLimit`. Esse sono impostate a valori di default e impongono ai container di non superare il numero di 1000 processi al loro interno e di non superare il vincolo di utilizzo della memoria RAM di 1 073 741 824 B, ossia 1 GiB. Anche in questo caso è opportuno che la limitazione delle risorse a disposizione dei vNSF Component della vNSF Reverse Proxy sia decisa dal vNSFO o dall'NFVI PoP, valutando il numero di istanze in esecuzione e l'hardware a disposizione.

5.4 vNSF Reverse Proxy: Reverse Proxy con WAF

Nella parte finale della Sezione 3.9, in seguito a diversi test, si è asserito come la combinazione httpd + ModSecurity sia la più idonea al contesto vNSF, per le sue prestazioni e il modello di rilevamento negativo, il quale le consente di essere immediatamente reattiva alle minacce appena avviata. Dopo questa considerazione, risulta naturale asserire che il Reverse Proxy con WAF della vNSF Reverse Proxy descritta in Figura 4.4 sia implementato con un container Docker che esegue al suo interno un'istanza di httpd con ModSecurity.

Sebbene il Reverse Proxy con WAF non sia il primo componente che interviene nell'avvio della vNSF Reverse Proxy, è trattato per primo. Difatti, dalla sua struttura dipende l'implementazione del Traduttore MSPL e del Collettore Log. Pertanto, in questa sezione si analizzano alcuni aspetti tecnici sia di httpd che di ModSecurity finora trascurati, al fine di facilitare la comprensione della Sezione 5.5 e della Sezione 5.6.

5.4.1 Virtual Host in httpd

Nella configurazione di httpd è di fondamentale importanza il concetto di Virtual Host. Nella documentazione ufficiale [70, Apache Virtual Host documentation], si usa tale termine per indicare

la possibilità di servire i contenuti di più siti web all'interno dello stesso host in modo completamente trasparente all'utente. Lo smistamento delle richieste entranti tra i diversi Virtual Host può avvenire sulla base dell'indirizzo IP o del nome di dominio richiesto.

Nel caso della vNSF Reverse Proxy, httpd utilizza il concetto di Virtual Host per poter inoltrare ogni richiesta entrante verso l'Origin Server destinatario. Tale aspetto è di notevole importanza poiché il processo di traduzione della MSPL necessita di avere conoscenza del formato dei file di configurazione. I Virtual Host in httpd sono descritti mediante direttive all'interno di un tag XML. Pertanto, è possibile anticipare che il Traduttore MSPL si occupa di tradurre configurazioni MSPL in file comprensibili da httpd.

5.4.2 Sintassi delle regole di ModSecurity

La Sezione 3.5.1 introduce ModSecurity, evidenziando come esso si basi su un modello di rilevamento negativo e rule-based. Nella presente sezione si analizza in primo luogo la sintassi necessaria per la comprensione e scrittura di regole utilizzabili nella configurazione di ModSecurity.

Il formato di una regola con ModSecurity prevede l'impiego del prefisso `SecRule` seguito da quattro blocchi [97, Capitolo 5]:

```
SecRule VARIABLES OPERATOR [TRANSFORMATION_FUNCTIONS, ACTIONS]
```

Le `[]` segnalano argomenti opzionali. In caso di loro mancanza, è implicito l'utilizzo di valori di default stabiliti precedentemente.

La sezione `VARIABLES` indica le parti della transazione HTTP a cui la regola è applicata. In ogni regola, è possibile specificare una o più variabili e ciascuna di esse rappresenta prevalentemente specifiche porzioni del traffico in ingresso o uscita dall'Origin Server. Durante l'analisi, il contenuto salvato in una variabile è codificato con stringhe binarie. Tale scelta si rende necessaria dal momento che un attaccante può tentare di compromettere l'obiettivo anche mediante caratteri speciali o byte di qualsiasi significato.

Le variabili possono essere categorizzate in diversi modi. Ad esempio, esse possono essere distinte in base all'entità cui si riferiscono (es. server, client) o alla modalità di memorizzazione (es. collezioni volatili, persistenti). Al fine di facilitare la comprensione si riportano alcune variabili:

- `ARGS` indica le intestazioni di una richiesta HTTP;
- `ARGS_GET` e `ARGS_POST` rappresentano rispettivamente i parametri presenti nelle richieste GET e POST;
- `REQUEST_BODY` e `RESPONSE_BODY` identificano il corpo di una richiesta e di una risposta;
- `REMOTE_ADDR` e `REMOTE_PORT` indicano indirizzo IP e porta della sorgente di una richiesta.

Per una lista completa delle variabili si rimanda alla documentazione ufficiale del progetto [119, Variables].

La sezione `OPERATORS` specifica in che modo è analizzata una variabile. Nella maggior parte dei casi un operatore specifica un *match* con determinate espressioni regolari, sebbene sia possibile definirne di nuovi. A differenza delle variabili, ogni regola prevede la presenza di un solo operatore, introdotto dal carattere `@`.

Gli operatori possono essere catalogati in base al tipo di analisi [97, Capitolo 5]:

String Matching Operators. Effettuano il match di un input con una stringa es. `@beginsWith`.

Numerical Operators. Confrontano input numerici evitando espressioni regolari es. `@eq`.

Validation Operators. Validano un input secondo un formato es. `@validateSchema` per dati codificati in XML.

```
SecRule REQUEST_URI "admin-login.php" "id:'1234567', deny, status:403"
```

Figura 5.4. Regola per vietare l'accesso a uno specifico URI.

```
SecRule REQUEST_HEADERS:User-Agent "@pmFromFile scanners-user-agents.data" \
 "msg:'Found User-Agent associated with security scanner',\
  severity:'CRITICAL',\
  id:913100,\
  rev:'2',\
  phase:request,\
  block,\
  ... "
```

Figura 5.5. Estratto della regola 913100 del progetto OWASP ModSecurity CRS v3.0.

Miscellaneous Operators. Effettuano controlli più specifici es. `@inspectFile` per eseguire uno script per il controllo di un file in input.

Analogamente alle variabili, si rimanda alla documentazione ufficiale per una lista completa degli operatori a disposizione [119, Operators].

Le `TRANSFORMATION_FUNCTIONS` sono utilizzate per alterare dati in input prima che essi siano utilizzati da un operatore. Ad esempio, la funzione `escapeSeqDecode` decodifica le sequenze di escape in ANSI C, tra cui `\n` `\a` `\"` `\'`. Allo stesso modo, `replaceNulls` sostituisce il byte `NUL` con spazi (ASCII `0x20`). È possibile concatenare più funzioni di trasformazione prima di passare un input al controllo dell'operatore. Si consulti [119, Transformation functions] per approfondire.

La sezione `ACTIONS` raccoglie metadati e operazioni da effettuare qualora il controllo di una data regola risultasse positivo. Sono presenti diverse categorie di azioni, utilizzabili contemporaneamente. Tuttavia, al fine di non rallentare la trattazione, si considera esclusivamente il caso delle *disruptive action*.

Una disruptive action costringe ModSecurity ad agire immediatamente sull'esito di una transazione nel caso in cui una regola sia soddisfatta. Gli interventi possono essere di carattere negativo o positivo. Nel primo caso rientrano la negazione con pagina di errore (`deny`), ridirezione (`redirect`) o chiusura della connessione (`drop`). Nel secondo si ritrova l'immediata accettazione della transazione senza procedere con l'elaborazione di altre regole (`allow`).

Per consolidare quanto trattato in questa sezione si esaminano ora due esempi di regole. In primo luogo si consideri la stringa in Figura 5.4.

Tale regola indica di controllare l'identificatore della risorsa richiesta (`REQUEST_URI`) e in caso di corrispondenza con la stringa indicata procedere alla negazione della transazione (`deny`) con messaggio di errore 403 (`status:403`).

Si consideri ora la Figura 5.5, la quale rappresenta un estratto di una regola appartenente al progetto OWASP ModSecurity CRS v3.0 e identificata con `id:913100`.

Il suo impiego prevede di analizzare l'intestazione `User-Agent` nelle richieste HTTP alla ricerca di *vulnerability scanner*, ossia software adoperati per la scansione delle vulnerabilità di un'applicazione. L'operatore `@pmFromFile` permette di testare la corrispondenza dell'input con un elenco di tali software riportato su un file. L'esame della regola avviene durante la fase di analisi dell'header (`phase:request`) e in caso di positività si procede al blocco della transazione (`block`).

```
--9b7ae92d-A--
[29/Sep/2017:15:55:56 +0000] Wc5tDCg-jsEQ9XIxQYwtPAAAAAQ 192.168.45.7 53942
192.168.45.8 80
--9b7ae92d-B--
GET /public/index.html HTTP/1.1
User-Agent: curl/7.29.0
Host: app
Accept: */*

--9b7ae92d-F--
HTTP/1.1 403 Forbidden
Content-Length: 219
Content-Type: text/html; charset=iso-8859-1

--9b7ae92d-E--

--9b7ae92d-H--
Message: Access denied with code 403 (phase 2). IPmatch "192.168.45.7" matched
"192.168.45.7" at REMOTE_ADDR. [file "/opt/proxy-conf/default-site.conf"]
[line "36"] [id "4300001"] [msg "Detected Bad IP"]
Action: Intercepted (phase 2)
Apache-Handler: proxy-server
Stopwatch: 1506700556686673 993 (- - -)
Stopwatch2: 1506700556686673 993; combined=395, p1=88, p2=302, p3=0, p4=0,
p5=5, sr=14, sw=0, l=0, gc=0
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.7.3 (http://www.modsecurity.org/);
OWASP_CRS/2.2.9.
Server: Apache/2.4.6 (CentOS)
Engine-Mode: "ENABLED"

--9b7ae92d-Z--
```

Figura 5.6. Esempio Audit Log di ModSecurity.

5.4.3 Struttura di un Audit Log in ModSecurity

La struttura dell'Audit Log di ModSecurity si presenta segmentata in diverse sezioni, ognuna delle quali è delimitata da un identificativo univoco per la transazione e una lettera dell'alfabeto.

La suddivisione in parti differenti consente di aggregare informazioni in base alla loro semantica. Ad esempio, la sezione `B` riporta l'header HTTP della richiesta in esame e la sezione `F` l'header della corrispondente risposta. Pertanto, è possibile registrare all'interno del Audit Log il contenuto di un'intera transazione HTTP. Le sezioni `A` (intestazione del file) e `Z` (chiusura) delimitano il contenuto di un singolo rilevamento e sono obbligatorie. Al contrario, le altre sezioni possono essere rimosse opzionalmente a seconda delle specifiche.

A titolo di esempio si consideri la Figura 5.6. Essa riporta un tentativo di accesso ad una risorsa Web da parte di un client avente indirizzo IP contrassegnato come pericoloso. Dalla sezione `A` si evince l'indirizzo IP e la porta sia della sorgente (`192.168.45.7 54942`) che della destinazione (`192.168.45.8 80`). I segmenti `B` ed `F` riportano le intestazioni della richiesta HTTP e della risposta. Il segmento `H` indica, invece, la ragione del rilevamento (`msg`) e il riferimento all'identificativo della regola (`id`).

5.5 vNSF Reverse Proxy: Traduttore MSPL

Nella Sezione 4.3.3 è stato affermato che il ruolo del Traduttore MSPL si concretizza nella traduzione del file in linguaggio MSPL salvato dal vNSF Controller nel MSPL Volume. Completata l'operazione, esso si arresta automaticamente.

Dal punto di vista implementativo, il Traduttore MSPL è un container in cui si esegue un software di traduzione da MSPL in configurazioni utilizzabili da httpd. Tale software prende il nome di *mspltranslator*.

Nel corso della presente sezione si riprende, in primo luogo, la trattazione sulla definizione di policy in MSPL focalizzando l'attenzione sul caso della vNSF Reverse Proxy. Successivamente, si descrive l'architettura software del mspltranslator.

5.5.1 Implementazione MSPL

Nel corso della Sezione 4.3.2 è stato presentato il linguaggio MSPL senza porre accento sulla sua implementazione. Nel corso di questo lavoro di tesi si considera come specifica progettuale l'implementazione di policy in MSPL mediante il linguaggio XML. Quest'ultimo offre la possibilità di validare il contenuto di una configurazione con l'impiego di un opportuno *XML Schema*. Nell'implementazione attuale delle vNSF si fa riferimento ad un solo XML Schema per la validazione di qualsiasi policy in MSPL sia implementata in XML, denominato `mspl_schema.xsd`.

In Figura 5.7 è possibile esaminare un esempio di file MSPL per la configurazione di un Reverse Proxy con WAF. In questo momento, se ne considera la sola struttura generale, rimandando un'analisi più approfondita alla trattazione successiva. Ciononostante, a partire dal nome dei tag XML, è possibile distinguere sequenzialmente la configurazione dei seguenti aspetti:

- indirizzo, porta e nome di dominio con cui la vNSF distingue il flusso entrante tra i vari Origin Server;
- path richiesto su un Reverse Proxy e traduzione con path su un Origin Server;
- configurazione SSL;
- configurazione di livello applicativo.

Tra gli aspetti rientranti in quest'ultima categoria vi sono:

- abilitazione di un WAF;
- abilitazione dell'analisi del corpo di una richiesta/risposta HTTP;
- IP bloccati e pagine sottoposte a Virtual Patch;
- pattern di attacco per cui si richiede l'intervento del WAF.

È di fondamentale importanza che si comprenda che ogni file MSPL si limita a descrivere esclusivamente policy di sicurezza. Pertanto, qualsiasi file MSPL non ha alcun legame con l'implementazione software. Ad esempio, nel caso della vNSF Reverse Proxy, il file MSPL non è in alcun modo collegato all'utilizzo di httpd + ModSecurity come Reverse Proxy con WAF. Dunque, vi è la possibilità di utilizzare lo stesso file MSPL con qualsiasi software in grado di applicare le direttive in esso contenute. Al contrario, per ogni software che si intende utilizzare sarà necessario un componente equivalente al Traduttore MSPL per la traduzione di policy in file di configurazione da esso comprensibili.

```
<?xml version="1.0" encoding="UTF-8"?>
<mspl-set xmlns="http://security.polito.it/shield/mspl"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://security.polito.it/shield/mspl
 mspl_schema_mod.xsd">
  <it-resource id="vNSF5">
    <configuration xsi:type="proxy-configuration">
      <rule>
        <source-address>*</source-address>
        <source-port>80</source-port>
        <destination-domain>www.example.com</destination-domain>
        <paths>
          <path>
            <path-dir>/site1</path-dir>
            <destination-address>192.168.45.3</destination-address>
            <destination-port>80</destination-port>
            <destination-path>/</destination-path>
            <destination-protocol>http</destination-protocol>
            <ssl>
              <status>NOTENABLED</status>
            </ssl>
            <condition>
              <application-layer-condition>
                <status>ENABLED</status>
                <default-response>403</default-response>
                <request-body-inspection>ENABLED</request-body-inspection>
                <response-body-inspection>NOTENABLED</response-body-inspection>
                <block-ip>192.168.45.7</block-ip>
                <restrict>broken_page.php</restrict>
                <pattern>
                  <name>XSS</name>
                  <status>ENABLED</status>
                </pattern>
              </application-layer-condition>
            </condition>
          </path>
        </paths>
      </rule>
    </configuration>
  </it-resource>
</mspl-set>
```

Figura 5.7. Esempio di MSPL per Reverse Proxy e WAF.

5.5.2 mspltranslator

Nell'introduzione alla Sezione 5.5 è stato asserito che la funzione del Traduttore MSPL è espletata all'interno del container da un software denominato mspltranslator. Analogamente al vNSF Controller, il mspltranslator è scritto in linguaggio Go e presenta un'architettura con un singolo thread.

L'esecuzione del mspltranslator richiede tre parametri in input: la configurazione in MSPL che si intende tradurre, il file mspl_schema.xsd e un *template di configurazione*. Quest'ultimo è un file contenente dei placeholder che aumenta la portabilità del mspltranslator consentendogli di

essere impiegato per altre implementazioni della vNSF Reverse Proxy. Si invita alla consultazione dell'Appendice B.3.8 per approfondire.

Lo sviluppo e l'aggiornamento del mspltranslator è determinato principalmente dall'evoluzione di due fattori: l'mspl_schema.xsd e il software per cui si propone di scrivere la traduzione, httpd nel caso della vNSF Reverse Proxy. Nel primo caso, è indispensabile che le strutture dati utilizzate da mspltranslator siano coerenti con i tipi di dato presenti nel mspl_schema.xsd. In mancanza di allineamento, è altamente probabile che si manifestino errori durante l'esecuzione. Nel secondo caso, è necessario che il mspltranslator conosca le direttive del software destinatario della sua traduzione. Pertanto, in caso di aggiornamento di quest'ultimo, è necessario verificare che non siano prodotte configurazioni di basso livello non più interpretabili o supportate.

Il ciclo di esecuzione del mspltranslator prevede l'esecuzione delle seguenti funzioni:

1. validazione della configurazione MSPL con il mspl_schema.xsd;
2. salvataggio dei dati decodificati in struct Go;
3. inserimento delle regole ModSecurity;
4. scrittura della configurazione di basso livello a partire dal template fornito in input.

5.5.3 Definizione degli Origin Server in httpd con mspltranslator

La struttura della configurazione in MSPL riportata in Figura 5.7 permette di impostare policy di sicurezza in maniera capillare. Tralasciando i tag XML presenti nell'intestazione, non rilevanti nelle impostazioni di sicurezza, la configurazione della vNSF Reverse Proxy inizia con il tag `<configuration>`. Al suo interno, è presente un vettore di `<rule>`.

Nell'implementazione attuale della vNSF Reverse Proxy vi è una corrispondenza biunivoca tra i tag `<rule>` e gli Origin Server. Nello specifico, ogni tag di questo tipo rappresenta le impostazioni di un singolo Virtual Host di httpd. Pertanto, è possibile, mediante la stessa istanza di una vNSF, proteggere in modo differente più Virtual Host. Tale caratteristica consente all'utente finale di scegliere il grado di protezione a seconda delle necessità di ogni Origin Server.

Per ogni Virtual Host, vi è la possibilità di specificare controlli differenti per ogni `<path>` dell'Origin Server. Ad esempio, un utente può richiedere una maggiore protezione per la directory `/private` all'interno del suo web server rispetto alla directory `/public`. Pertanto, le policy di sicurezza rappresentate all'interno dei tag `<application-layer-condition>` hanno la granularità della singola directory all'interno del Origin Server.

5.5.4 Scrittura delle regole di ModSecurity con mspltranslator

Il mspltranslator restituisce una configurazione che comprende regole di ModSecurity sia appartenenti al OWASP ModSecurity CRS sia sviluppate nel contesto di questo lavoro di tesi. Ai fini della trattazione, si considerano solo i due casi appartenenti alla seconda categoria. Si premette che il mspltranslator può essere esteso alla scrittura di ulteriori regole per ModSecurity, contestualmente all'aumento di complessità delle policy nel mspl_schema.xsd.

Negazione richieste provenienti da un dato indirizzo IP

Il tag XML `<block-ip>`, presente all'interno della configurazione in MSPL riportata in Figura 5.7, consente di specificare un indirizzo IP a cui vietare l'accesso ad un determinato `<path>`. L'utilizzo di più tag di questo tipo permette la creazione di una blacklist. Nelle implementazioni future, non si esclude la possibilità di estendere il mspl_schema.xsd con un tag `<allow-ip>` che consenta di compiere l'operazione opposta, ossia la creazione di una whitelist.

All'interno del mspltranslator la creazione di una blacklist comprensibile da ModSecurity avviene mediante regole aventi il formato riportato in Figura 5.8.

```
SecRule REMOTE_ADDR "@ipMatch {{.IP}}" "id:'{{.RuleID}}',deny,
    msg:'Detected Bad IP'"
```

Figura 5.8. Regola per la negazione delle richieste provenienti da un dato indirizzo IP.

```
SecRule REQUEST_URI "{{.Path}}" "id:'{{.RuleID}}',phase:1,deny,msg:'Attempt to
access restricted path'"
```

Figura 5.9. Regola per la negazione delle richieste indirizzate ad uno specifico URI.

L'operatore `@ipMatch` è specifico per effettuare un rapido confronto tra indirizzi IP. Si nota la presenza di due placeholder: `{{.IP}}` e `{{.RuleID}}`: il primo è prelevato da ogni tag `<block-ip>`, mentre il secondo è assegnato da mspltranslator con una strategia di conteggio interna. L'azione `deny` specificata comporta la negazione della richiesta con messaggio di errore, definito nelle impostazioni di default.

Negazione richieste indirizzate ad uno specifico URI

Il tag `<restrict>` consente di negare l'accesso ad un determinato URI all'interno di un `<path>`. I casi che possono richiedere l'applicazione di tale regola sono diversi. Tra questi si possono includere il Virtual Patching di una pagina contenente vulnerabilità e la negazione di accesso remoto a determinate pagine di amministrazione del Origin Server.

All'interno del mspltranslator il formato delle regole che svolgono tale compito è riportato in Figura 5.9.

Sebbene non indicato esplicitamente, l'operatore utilizzato rientra nella categoria String Matching Operators (Sezione 5.4.2). L'URI è estratto dal tag `<restrict>` e inserito all'interno del placeholder `{{.Path}}`. Analogamente al caso precedente, mspltranslator si occupa internamente dell'assegnazione dei `{{.RuleID}}`. Qualora la regola sia soddisfatta, anche in questo caso si procede alla negazione della richiesta inviando all'utente un messaggio di errore.

Rilevamento dei comuni pattern di attacco

Il tag `<pattern>` all'interno delle configurazioni in MSPL permette di richiedere la protezione, con ModSecurity, da specifici pattern di attacco. Nello specifico, esso presenta al suo interno un tag `<name>` per la denominazione del tipo di attacco e un flag `<status>` per richiederne o meno il bloccaggio. Nella Figura 5.7 è possibile osservare come sia stata richiesta la protezione da attacchi di tipo XSS.

La traduzione della direttiva in MSPL all'interno di mspltranslator è effettuata mediante l'ausilio di una mappa. Essa consente di associare ad un determinato pattern di attacco una lista di file, contenenti regole per il rilevamento. È indispensabile precisare che tali file appartengono al già citato progetto OWASP ModSecurity CRS nella versione 3.0. Si consiglia di consultare l'Appendice B.3.1 per approfondire.

5.6 vNSF Reverse Proxy: Collettore Log

Nella Sezione 4.3.3 è stato anticipato che il ruolo del Collettore Log consiste nell'aggregazione e manipolazione del contenuto dell'Audit Log Volume affinché questo sia comprensibile dal DARE.

Nell'implementazione attuale della vNSF Reverse Proxy, il Collettore Log è stato realizzato mediante un container che esegue al suo interno il software Logstash [120]. Logstash è un software open source in grado di ricevere dati da diverse sorgenti, aggregarli in un formato specifico e inviarli ad un utilizzatore. Esso si dimostra particolarmente idoneo al caso della vNSF Reverse Proxy, poiché la sua versatilità gli permette di essere utilizzato per comunicare con Reverse Proxy con WAF sia tramite volumi sia interfacce di rete. Pertanto, un possibile aggiornamento del Reverse Proxy con WAF che preveda di cambiare la modalità di interazione non influisce sull'implementazione del Collettore Log, rendendoli totalmente disaccoppiati.

Il processo di elaborazione di Logstash prende il nome di *pipeline* ed è costituito da tre componenti che intervengono sequenzialmente:

- Input, rappresentano le sorgenti di dato: file, database, socket ecc.;
- Filter, effettuano la manipolazione dei dati raccolti in Input es. estrazione di parametri da stringhe;
- Output, inviano i dati manipolati ad utilizzatori o altri database.

Ciascuno dei succitati componenti utilizza al suo interno dei *plugin* per svolgere il suo compito. Ad esempio, vi sono plugin di Input per l'interfacciamento con file, socket, database ecc. Allo stesso modo, vi sono plugin di Filter per estrarre contenuti da stringhe, manipolarne il valore, decifrarlo ecc. È opportuno sottolineare la possibilità di programmare nuovi plugin qualora nessuno di quelli già offerti risulti idoneo al caso d'uso.

Nella Figura 5.10 vi è una rappresentazione della pipeline di Logstash nel caso della vNSF Reverse Proxy. Nel blocco centrale sono visibili le tre componenti succitate.

Nel punto 1 è possibile esaminare la configurazione di un Input. Il plugin *file* utilizzato in questo caso permette l'acquisizione di contenuti da file. Nel caso dell'implementazione della vNSF Reverse Proxy, i file in oggetto sono parte del Audit Log di ModSecurity, salvati all'interno del Audit Log Volume. Difatti, è possibile osservare la direttiva `path` all'interno della configurazione che punta alla directory in cui è salvato il Volume (`AUDITLOGPATH`).

Nel punto 2 è rappresentata la configurazione di un Filter di Logstash. È stato utilizzato il plugin *grok* [121], il quale permette di strutturare dati in modo tale che possano essere successivamente interrogati.

I punti 3 e 4 illustrano la situazione attuale e gli sviluppi futuri dell'implementazione della vNSF Reverse Proxy. Attualmente, il risultato dell'elaborazione di Logstash è indirizzato ad un quarto Volume, denominato Output Log Volume e già citato nella Sezione 5.3.1. Gli sviluppi futuri prevedono l'implementazione di un *DARE_plugin* in grado di creare un interfacciamento diretto tra Logstash e il DARE.

Al fine di chiarire quanto esposto finora si consideri il seguente esempio. Siano:

- C, un generico client con indirizzo IP 192.168.45.9;
- S, un generico Origin Server protetto dalla vNSF Reverse Proxy con hostname *webapp*;
- RP, la vNSF Reverse Proxy implementata con httpd e ModSecurity.

Si supponga che C tenti di effettuare una richiesta illecita verso il server S, ignaro della presenza di RP. RP, rilevata la minaccia, produce un Audit Log diviso in diverse sezioni, come già osservato nella Sezione 5.4.3.

Si consideri come Audit Log di RP il frammento riportato in Figura 5.11. Ogni sua porzione è sottoposta ad un diverso Filter di Logstash. A tal proposito, si consideri il contenuto della Sezione A dell'Audit Log, la quale identifica il client che ha effettuato la richiesta ritenuta illecita.

Essa può essere elaborata dal Filter avente la configurazione simile a quella presente nel progetto Github *bitsofinfo/logstash-modsecurity* [122], rappresentata in Figura 5.12.

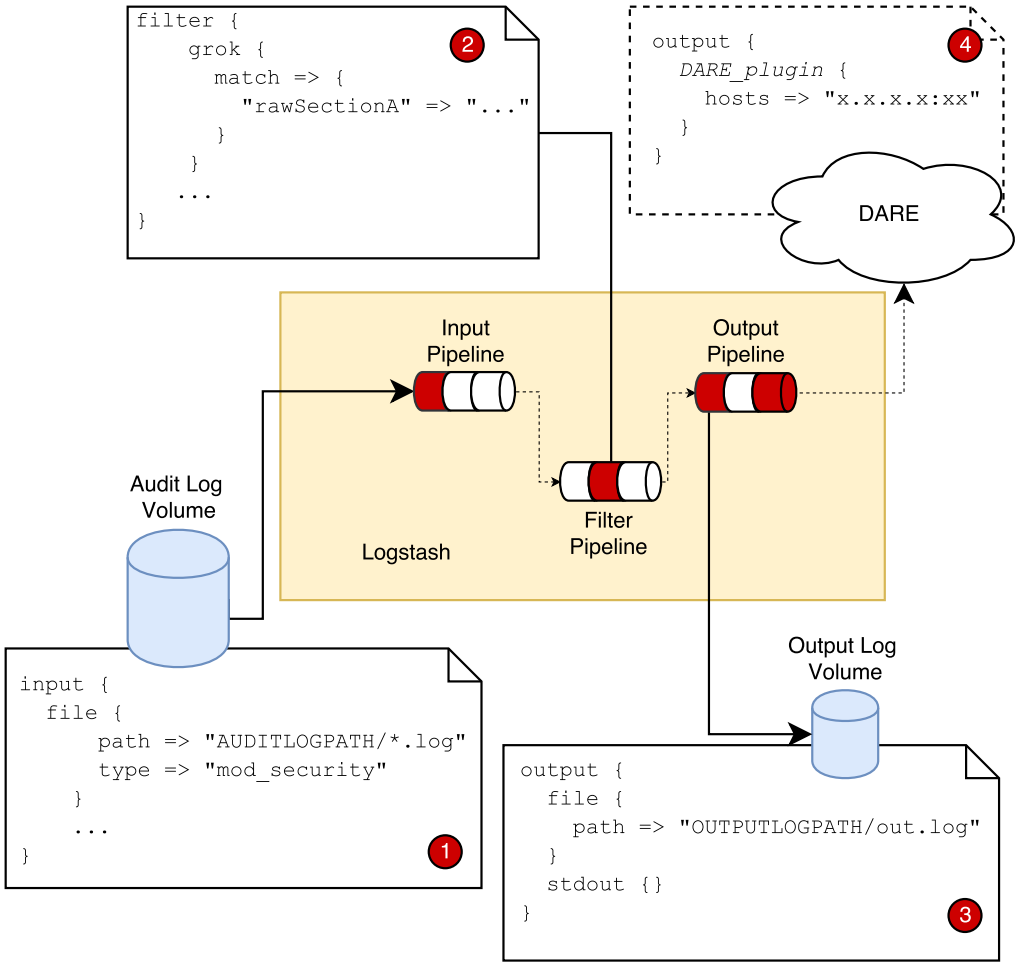

Figura 5.10. Pipeline in Logstash.

```
--13d29b59-A--
[22/Oct/2017:16:56:34 +0000] WezNwr3AQ2zl@5uWAQExowAAAAI 192.168.45.9 53948
192.168.45.8 80
--13d29b59-B--
GET /public/index.html HTTP/1.1
User-Agent: curl/7.29.0
Host: webapp
Accept: */*

--13d29b59-F--
HTTP/1.1 403 Forbidden
Content-Length: 219
Content-Type: text/html; charset=iso-8859-1

...
```

Figura 5.11. Audit Log di RP in seguito ad una richiesta illecita.

```
filter {
  if [type] == "mod_security" {
    grok {
      match => {
        "rawSectionA" =>
        "\[(?<modsec_timestamp>%{MONTHDAY}/%{MONTH}/%{YEAR}:%{TIME}
         [-\+]{1,2}%{INT})\] %{DATA:uniqueId} %{IP:sourceIp} %{INT:sourcePort}
         %{IP:destIp} %{INT:destPort}"
      }
    }
    geoip {
      source => "sourceIp"
    }
  }
}
```

Figura 5.12. Filter per l'elaborazione dell'Audit Log.

```
{
    "modsec_timestamp" => "22/Oct/2017:16:56:34 +0000"
    "uniqueId" => "WezUj2BV3HGGzSaj6J0A3gAAAAQ",
    "sourceIp" => "192.168.45.9",
    "sourcePort" => "54120",
    "destIp" => "192.168.45.8",
    "destPort" => "80",
    "geoip" => {},
    "rawSectionA" => "[22/Oct/2017:16:56:34 +0000] WezNwr3AQ2zl@5uWAQExowAAAAI
    192.168.45.9 53948 192.168.45.8 80",
    ...
    (content from other Filters)
    ...
}
```

Figura 5.13. Esempio di output di Logstash.

In essa si nota la presenza di due plugin: *grok* e *geoip*. Il primo esamina il contenuto della sezione A e lo salva in opportune variabili con un determinato tipo. Ad esempio, le variabili `sourcePort` e `destPort`, rappresentanti porta sorgente e destinazione, sono salvate come dati di tipo `INT` all'interno di Logstash. Il secondo, permette di localizzare geograficamente l'indirizzo IP della richiesta sorgente. Nel caso in esame, esso non si dimostra particolarmente utile dal momento che C utilizza un indirizzo privato.

Una volta completata l'azione dei Filter il risultato è da ritenersi simile a quello in Figura 5.13.

Risulta evidente come questo possa essere facilmente memorizzato in un database a documenti e interrogabile da un software come il DARE.

Capitolo 6

Collaudo

6.1 Overview

Nel Capitolo 5 è stata proposta un'implementazione della vNSF Reverse Proxy, analizzandone i software che la compongono e la loro configurazione. Tale implementazione costituisce un *Proof of Concept* (PoC), il quale può essere utilizzato come riferimento per successive implementazioni della stessa vNSF o di altri tipi, studiati per funzionalità differenti.

Nel presente capitolo si esaminano due diversi scenari di collaudo. Nel primo si valutano le differenze, in termini di prestazioni, tra diversi possibili utilizzi di una soluzione Reverse Proxy con WAF costituita da httpd e ModSecurity. Nel secondo scenario si valuta una possibile estensione dell'architettura di deployment della vNSF Reverse Proxy mediante un Load Balancer. In questo caso, si valutano differenze di prestazioni tra un'implementazione provvista e una non dotata di questa estensione.

6.2 Collaudo vNSF Reverse Proxy

La presente sezione si pone l'obiettivo di confrontare le prestazioni della combinazione di Reverse Proxy con WAF, costituita da httpd e ModSecurity, in tre scenari differenti, denominati *caso Standalone*, *caso Docker* e *caso VM*.

Il caso Standalone prevede l'utilizzo di httpd e ModSecurity direttamente all'interno di un sistema host, senza l'ausilio di alcuna piattaforma di virtualizzazione hardware o leggera. Il caso Docker consiste nell'impiego di container all'interno dell'host, secondo l'implementazione della vNSF Reverse Proxy descritta nel corso del Capitolo 5. Infine, il caso VM prevede l'impiego di httpd con ModSecurity all'interno di una VM in esecuzione sul sistema host.

Nella trattazione seguente si esaminano ordinatamente: l'architettura dell'ambiente di collaudo, le modalità con cui sono stati valutati gli scenari ed i risultati ottenuti.

6.2.1 Ambiente di collaudo

Al fine di poter meglio distinguere il ruolo delle parti coinvolte, sono stati individuati tre attori all'interno dell'ambiente di collaudo: Client, Origin Server e Reverse Proxy con WAF. Quest'ultimo, come riportato in Figura 6.1, può essere realizzato in diversi modi all'interno di un host, a seconda che si tratti del caso Standalone, del caso Docker o del caso VM.

L'architettura utilizzata per il Client è la seguente:

- Intel Core i7-4510U CPU @ 2.00GHz, 2 core;

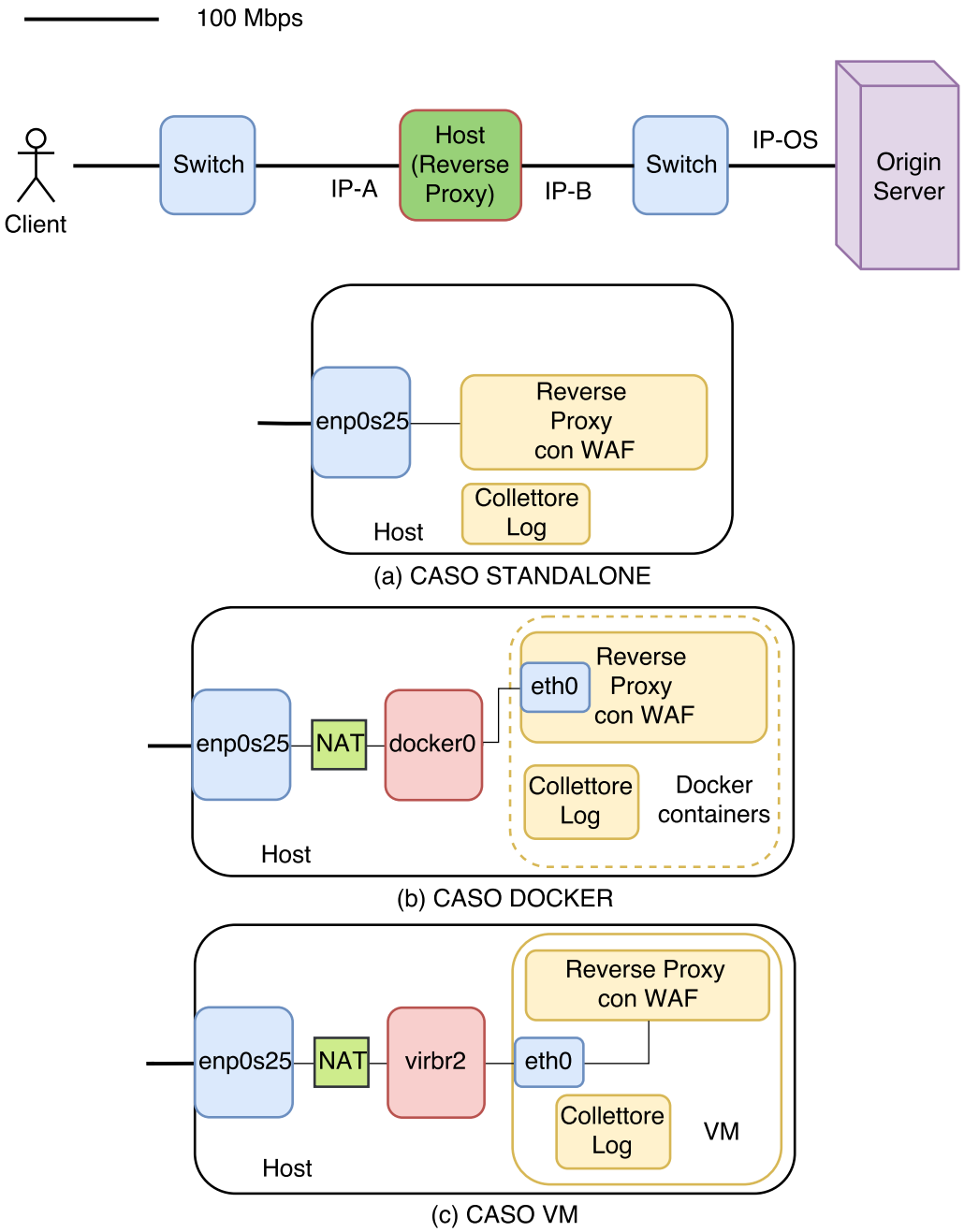

Figura 6.1. Architettura per il collaudo della vNSF Reverse Proxy.

- 12 GiB di RAM DDR3;
- OS Debian 9 Stretch 64 bit con kernel Linux 4.9.0-3-amd64;
- Apache HTTP server benchmarking tool `ab`.

Per le funzionalità di Reverse Proxy e WAF, in ognuno dei suddetti casi, è stato utilizzato un host avente la seguente configurazione:

- Intel Core i5-5300U CPU @ 2.30GHz, 2 core;
- 16 GiB di RAM DDR3;

- OS CentOS 7 64 bit con kernel Linux 3.10.0

A partire da questa configurazione, sono stati aggiunti componenti software a seconda del caso in esame.

Nel caso Standalone, al Sistema Operativo dell'host sono stati aggiunti httpd versione 2.4.6 e ModSecurity versione 2.7.3-5.el7.

Nel caso Docker, a partire dall'installazione base di CentOS 7 è stato aggiunto Docker nella versione 17.06.0-ce. Inoltre, è stato utilizzato il software `iptables` [123], uno strumento per la gestione dei pacchetti su sistemi Linux. In particolare, può essere utilizzato in veste di Firewall o NAT, per configurazioni di diversa difficoltà. Nel caso in esame, `iptables` consente di effettuare l'instradamento dei pacchetti in entrata nell'host verso i container mediante regole analoghe a quelle riportate in Figura 6.2.

Nel caso VM, il sistema host è stato dotato di una VM avente la seguente configurazione:

- 1 GiB di RAM;
- 1 vCPU;
- OS Centos 7 64 bit con kernel Linux 3.10.0;
- httpd versione 2.4.6
- ModSecurity versione 2.7.3-5.el7

Inoltre, è stato utilizzato il software `iptables` per consentire l'instradamento dei pacchetti in entrata nell'host verso la VM.

È necessario segnalare che il Reverse Proxy con WAF del caso Docker esegue le stesse versioni di httpd e ModSecurity utilizzate nei casi Standalone e VM.

L'architettura utilizzata per l'Origin Server consiste in:

- Intel Core 2 CPU 6400 @ 2.13GHz, 2 core;
- 2 GiB di RAM DDR2;
- OS Centos 7 64 bit con kernel Linux 3.10.0;
- httpd nella versione 2.4.6.

È opportuno precisare che l'architettura hardware in dotazione al Client e all'Origin Server è qui riportata solo per consentire l'inquadramento dell'ambiente di collaudo. Difatti, nel deployment della vNSF Reverse Proxy non è possibile fare alcuna assunzione circa il tipo di architettura in dotazione ai Client o agli Origin Server da essa protetti. Di questi ultimi, ad esempio, non è possibile sapere a priori della loro configurazione e se siano in esecuzione o meno su istanze virtualizzate.

```
iptables -t nat -I PREROUTING -p tcp -i enp0s25 --dport 80 -j DNAT \
--to IP:80
iptables -A FORWARD -i enp0s25 -o vibr0 -p tcp --dport 80 -j ACCEPT
```

Figura 6.2. Esempio regole `iptables` per la traduzione delle richieste dirette all'indirizzo IP-A.

6.2.2 Modalità di esecuzione

Nella Figura 6.1 è possibile osservare l'interazione tra i tre attori dell'ambiente di collaudo.

Il Client si occupa di espletare richieste indirizzate ad un nome di dominio es. `www.example.org`. Al fine di evitare latenze dovute alla risoluzione di tale nome è stata prevista una risoluzione statica nel file `/etc/hosts` ad un dato indirizzo IP, da qui detto *IP-A*. Il sistema host è in attesa di connessioni per l'indirizzo IP-A sull'interfaccia `enp0s25`. L'elaborazione di una richiesta in entrata dipende dal tipo di caso in esame.

caso Standalone

La richiesta è ricevuta dall'istanza di httpd direttamente in ascolto sulla porta 80/TCP del sistema host. Se non ritenuta potenzialmente pericolosa, si procede all'inoltro attraverso l'interfaccia `enp0s25` verso l'Origin Server in ascolto sull'indirizzo IP-OS.

caso Docker

Una richiesta ricevuta è inoltrata, grazie all'ausilio delle regole di traduzione di `iptables`, sul Virtual Bridge `docker0`. Tali regole sono definite direttamente dal Docker Daemon nel momento in cui si specifica un port mapping. Successivamente, i pacchetti sono indirizzati dal Virtual Bridge all'interfaccia `eth0` presente all'interno del container Reverse Proxy con WAF. Quest'ultimo prevede al suo interno un'istanza di httpd con ModSecurity in ascolto sulla porta 80/TCP. Una volta elaborata, se non ritenuta potenzialmente pericolosa, la richiesta di partenza del Client è inoltrata all'Origin Server in ascolto sull'indirizzo IP-OS.

caso VM

Analogamente al caso precedente, una richiesta ricevuta è inoltrata, mediante regole di `iptables`, sul Virtual Bridge `virbr2`. Tali regole sono state definite appositamente, secondo una modalità analoga a quella riportata in Figura 6.2. Successivamente, i pacchetti sono indirizzati dal Virtual Bridge all'interfaccia `eth0` della VM in esecuzione sull'host. All'interno della VM, un'istanza di httpd con ModSecurity, in ascolto sulla porta 80/TCP, esamina la richiesta e procede, a meno di rilevamenti sospetti, all'inoltro verso l'Origin Server in ascolto sull'indirizzo IP-OS.

6.2.3 Risultati

Il collaudo dei due casi di studio è avvenuto effettuando un flusso di richieste `HTTP/1.1` della durata di 120 s, con metodo `GET` e intestazione `Connection: keep-alive`, per una risorsa `index.html` della dimensione di 40 KiB. Inoltre, mediante il software `ab` nella versione 2.3, è stata simulata al tempo stesso una concorrenza di 50, 100 e 250 utenti concorrenti.

L'Origin Server elabora le richieste senza alcuna indicazione aggiuntiva circa il caso d'uso che ha generato la risposta. Difatti, qualsiasi richiesta entrante presenta indirizzo sorgente IP-A.

Nella Figura 6.3 sono stati riportati i risultati del collaudo in termini di throughput.

In primo luogo, si osserva come le prestazioni del caso Standalone e del caso Docker differiscano minimamente. Nello specifico, si osserva un calo di throughput del caso Docker rispetto al caso Standalone che varia dal 1,94%, con una concorrenza di 50 utenti, al 3.24%, con 250 utenti. Al contrario, il caso VM presenta delle prestazioni nettamente inferiori rispetto agli altri due casi in esame. In particolare, si osserva una diminuzione del throughput rispetto al caso Standalone compresa tra il 28,85%, con una concorrenza di 250 utenti, e del 36,73%, con una concorrenza di 50 utenti.

Il motivo di una tale differenza di prestazioni tra i due scenari virtualizzati, ossia il caso Docker e il caso VM, può essere ricondotta alla necessità di attraversare, nel secondo caso, il Sistema

Operativo della VM prima di poter procedere all'esame della richiesta da parte di httpd. Al contrario, la condivisione del kernel favorisce le prestazioni del caso Docker, il quale presenta minime differenze di throughput rispetto al caso Standalone in tutti gli scenari.

In secondo luogo, si osserva un sensibile calo delle prestazioni all'aumentare della concorrenza. Ad esempio, nel caso Docker si osserva una diminuzione del 15,14% tra il throughput con una concorrenza di 50 utenti e il throughput con una concorrenza di 250 utenti. Per limitare il calo di prestazioni è opportuno optare per una distribuzione del carico computazionale tra più istanze dell'Origin Server, possibilmente tramite l'ausilio di un Load Balancer.

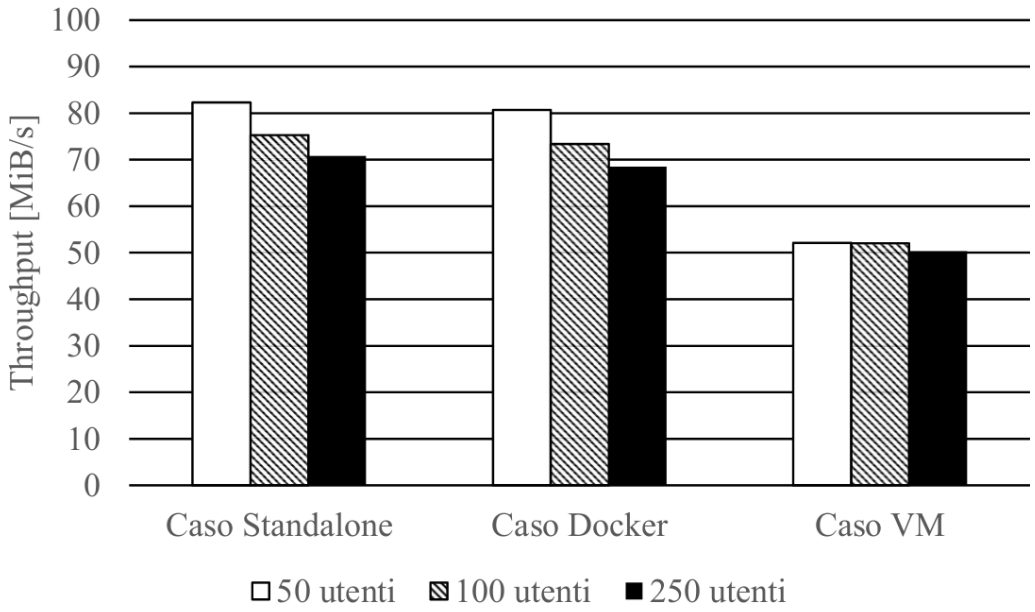

Figura 6.3. Throughput dei tre casi di studio nell'utilizzo di httpd con ModSecurity.

6.3 Collaudo vNSF Reverse Proxy con Load Balancer

Nella Sezione 4.5.3 è stata discussa l'introduzione di un Load Balancer nell'architettura della vNSF Reverse Proxy, supportata dall'utilizzo di Overlay Network (Sezione 4.5.2). Nello specifico, è stato visto come l'impiego di un Load Balancer possa aumentare il livello di Fault-Tolerance dell'infrastruttura complessiva e consentire la scalabilità orizzontale della vNSF Reverse Proxy.

Nella presente sezione si presenta una possibile configurazione di vNSF Reverse Proxy con Load Balancer, al fine di garantire una distribuzione del carico computazionale derivante dalle richieste dei Client su più istanze.

6.3.1 Premessa

Prima di procedere alla descrizione dell'ambiente di collaudo, è opportuno premettere quale sia il suo scopo. La presente sezione *non* si pone l'obiettivo di valutare le prestazioni di un Load Balancer. Difatti, analogamente a quanto visto nel Capitolo 3 per i Reverse Proxy, è possibile identificare diverse alternative presenti sul mercato. Essa propone, invece, un possibile utilizzo di un Load Balancer per supportare la distribuzione del carico.

Nella Sezione 6.2 è stato valutato il throughput di diversi possibili scenari di utilizzo della combinazione Reverse Proxy con WAF costituita da httpd con ModSecurity. In questa sezione si propone di collaudare la vNSF Reverse Proxy del caso Docker, ossia implementata mediante container, sia nel caso di una singola istanza, da qui denominato *caso vNSF*, sia nel caso di istanze multiple coordinate da un Load Balancer, identificato come *caso vNSF + Load Balancer*.

6.3.2 Ambiente di collaudo

Analogamente a quanto riportato nella Sezione 6.2.2, sono stati individuati tre attori all'interno dell'ambiente di collaudo: Client, Origin Server e ambiente di produzione.

L'architettura utilizzata per il Client è la stessa utilizzata nel precedente scenario di collaudo:

- Intel Core i7-4510U CPU @ 2.00GHz, 2 core;
- 12 GiB di RAM DDR3;
- OS Debian 9 Stretch 64 bit con kernel Linux 4.9.0-3-amd64;
- Apache HTTP server benchmarking tool `ab`.

L'architettura utilizzata per l'Origin Server è la seguente:

- Intel Core i5-5300U CPU @ 2.30GHz, 2 core;
- 16 GiB di RAM DDR3;
- OS CentOS 7 64 bit con kernel Linux 3.10.0

L'ambiente di produzione presenta un'architettura differente, a seconda che si tratti del caso vNSF o del caso vNSF + Load Balancer.

caso vNSF

L'ambiente di produzione del caso vNSF è rappresentato in Figura 6.4. Esso consiste in una sola istanza della vNSF Reverse Proxy, in esecuzione su un host avente la seguente considerazione:

- Intel Core 2 CPU 6400 @ 2.13GHz, 2 core;
- 2 GiB di RAM DDR2;

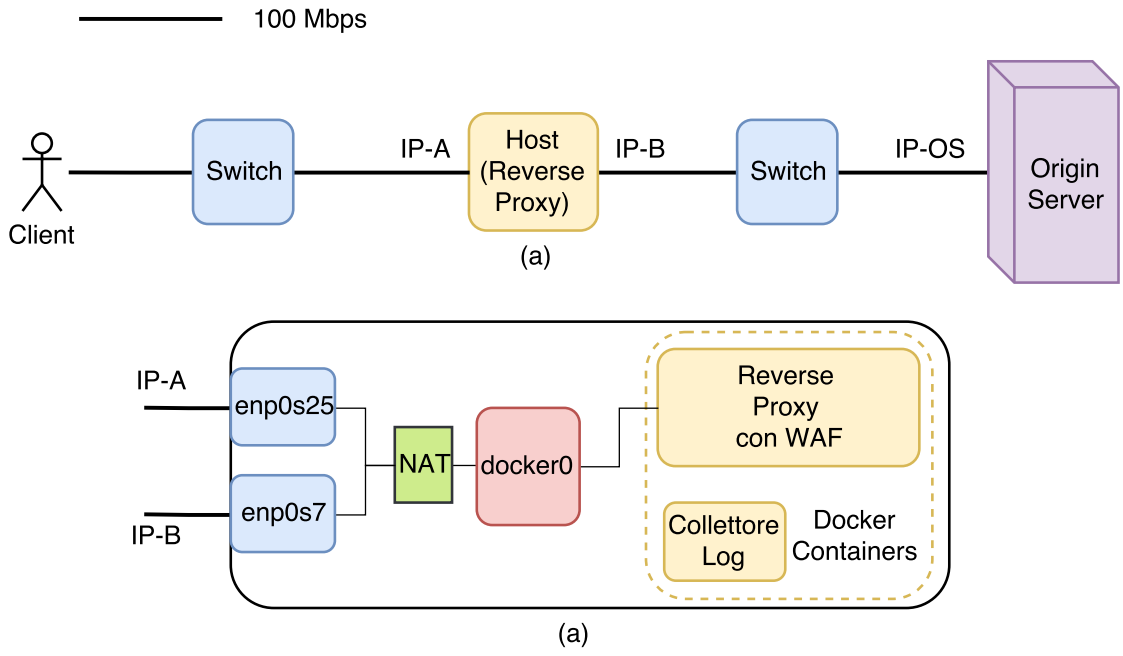

Figura 6.4. Architettura di collaudo del caso vNSF.

- OS Centos 7 64 bit con kernel Linux 3.10.0;
- Docker nella versione 17.06.0-ce.

Inoltre, esso è collegato a due switch differenti mediante link a 100 Mibit/s.

caso vNSF + Load Balancer

L'ambiente di produzione del caso vNSF + Load Balancer è illustrato in Figura 6.5. Esso conta

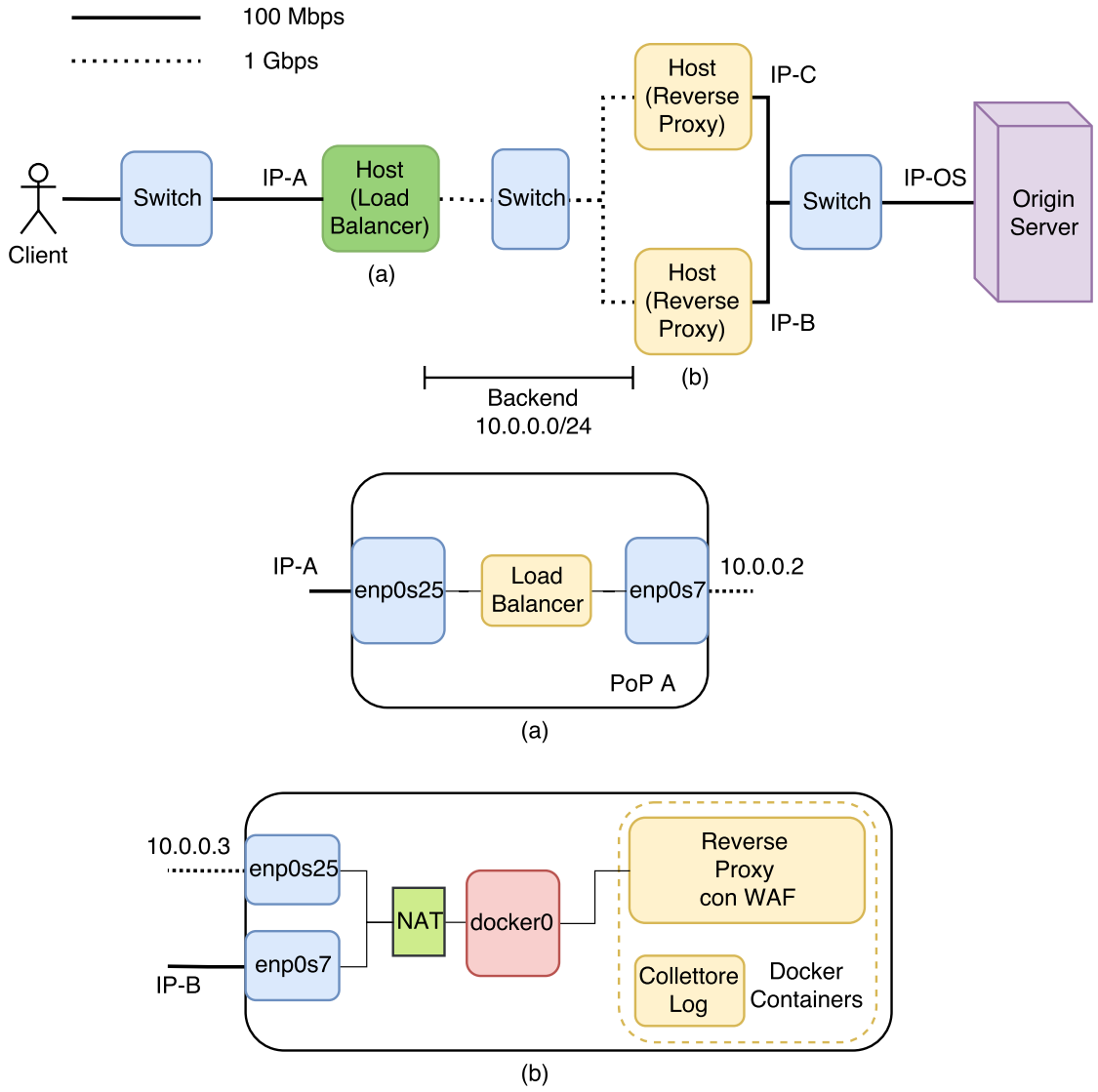

Figura 6.5. Architettura di collaudo del caso vNSF + Load Balancer.

tre host differenti, uno per la realizzazione del Load Balancer e due per l'esecuzione di due istanze differenti della vNSF Reverse Proxy, aventi la stessa dotazione hardware:

- Intel Core 2 CPU 6400 @ 2.13GHz, 2 core;
- 2 GiB di RAM DDR2;
- OS Centos 7 64 bit con kernel Linux 3.10.0.

Ognuno di essi presenta due interfacce di rete fisiche differenti. La prima consente la creazione di una rete di backend, costituita da link di 1 Gibit/s. La seconda consente l'interfacciamento del Load Balancer con il Client e dei Reverse Proxy con l'Origin Server, mediante link a 100 Mibit/s.

Per le operazioni di load balance è stata utilizzato HAProxy nella versione 1.5.18. Tale scelta non è da ritenersi in alcun modo vincolante.

6.3.3 Modalità di esecuzione

Lo scenario di collaudo prevede che il Client si occupi di espletare richieste indirizzate ad un nome di dominio es. `www.example.org`. Al fine di evitare latenze dovute alla risoluzione di tale nome, è stata prevista una risoluzione statica nel file `/etc/hosts` ad un dato indirizzo IP, da qui detto *IP-A*.

Il processo di elaborazione di una richiesta in entrata dipende dal tipo di caso in esame.

Nel caso vNSF, una richiesta da parte del Client è direttamente ricevuta dall'interfaccia fisica `enp0s25` (Figura 6.4). Da quest'ultima, mediante regole di `iptables` definite dal Docker Daemon, una richiesta è inoltrata al container Reverse Proxy con WAF in esecuzione sull'host. Se la richiesta è ritenuta lecita, quest'ultimo procede all'inoltro verso l'Origin Server. Nello specifico, una volta raggiunto il Virtual Bridge `docker0`, opportune regole di routing consentono l'instradamento sull'interfaccia fisica `enp0s7`

Nel caso vNSF + Load Balancer, una richiesta da parte del Client è ricevuta dall'interfaccia fisica `enp0s25` dell'host avente in esecuzione il Load Balancer (HAProxy nel caso in esame). Esso effettua lo smistamento sulle due istanze disponibili della vNSF Reverse Proxy operando a livello quattro dello stack TCP/IP e secondo una politica *round robin*, la quale prevede una distribuzione alternata del carico tra le varie istanze.

Una volta ricevuta la richiesta dall'interfaccia esposta sulla rete di backend, identificata dall'indirizzamento `10.0.0.0/24`, l'istanza selezionata della vNSF Reverse Proxy procede all'elaborazione della richiesta secondo le stesse modalità del caso vNSF.

6.3.4 Risultati

Il collaudo dei due casi di studio è avvenuto effettuando un flusso di richieste `HTTP/1.1` della durata di 120 s, con metodo `GET` e intestazione `Connection: keep-alive`, per una risorsa `index.html` della dimensione di 40 KiB. Inoltre, mediante il software `ab` nella versione 2.3, è stata simulata al tempo stesso una concorrenza di 50, 100 e 250 utenti concorrenti.

Nella Figura 6.6 è possibile osservare il risultato in termini di throughput.

È evidente come l'utilizzo di due istanze differenti della vNSF Reverse Proxy coordinate da un Load Balancer consenta un notevole aumento delle prestazioni dell'ambiente di produzione, di poco superiore al 70%. Non è possibile assistere ad un raddoppio totale delle prestazioni a causa dell'intervento del Load Balancer e della presenza di un link aggiuntivo nel percorso tra il Client e l'Origin Server.

Inoltre, l'inclusione del Load Balancer permette di aumentare il grado di Fault-Tolerance. Difatti, un possibile guasto all'host contenente la prima istanza della vNSF Reverse Proxy non causa la mancata protezione o l'isolamento dell'Origin Server.

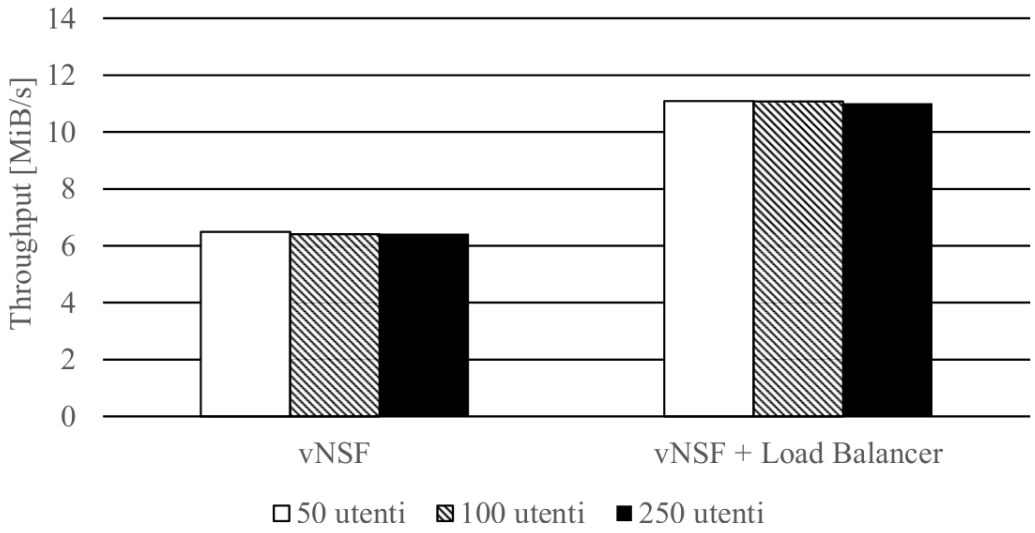

Figura 6.6. Throughput del caso vNSF e del caso vNSF + Load Balancer.

Capitolo 7

Conclusioni

Il presente elaborato considera contemporaneamente diverse tecnologie e diversi aspetti, cercando di utilizzarli congiuntamente nella progettazione e nell'implementazione della vNSF Reverse Proxy. A partire da questi, è possibile fare le seguenti considerazioni.

Innanzitutto, è evidente come la progettazione di una vNSF non possa essere condotta senza la conoscenza del tipo di virtualizzazione offerta nell'ambiente di deployment. In primo luogo, nel corso del Capitolo 2 è stato ampiamente dimostrato come l'impiego di container richieda maggiori considerazioni inerenti la sicurezza rispetto all'utilizzo di VM. Ad esempio, è necessario evitare che vi siano interazioni impreviste tra container. Ovvero, è opportuno limitare l'accesso alle risorse hardware per evitare possibili scenari di DoS. In secondo luogo, la virtualizzazione leggera basata su Application container prevede l'utilizzo di entità effimere e senza stati, a differenza di quanto accade nei Machine container e nelle VM.

In seguito, è stato visto come l'architettura di vNSF basate su tecniche di Virtualizzazione Leggera non possa essere ritenuta totalmente *compliant*, ossia conforme, agli standard ETSI citati nel corso della trattazione. Nello specifico, nella Sezione 4.3.3 è stato visto come non sia possibile adottare, in un contesto basato su container, le stesse modalità di management e orchestrazione proposte per le VNF. Si è proposto di colmare tale divergenza mediante il vNSF Controller, il quale svincola un orchestratore dalla necessità di interfacciarsi con il Docker Daemon in esecuzione su un NFVI PoP. In altri termini, si offre ad un'entità di management la possibilità di avere una panoramica dello stato di una vNSF senza dover interrogare ogni singolo vNSF Component che la costituisce.

Successivamente, la struttura modulare proposta per le vNSF consente sia di avere un controllo capillare sulle singole funzionalità sia di aggiornarle senza dover modificare l'architettura preesistente. Nel primo caso, si è visto come interrogando il vNSF Controller si possa evincere lo stato dell'elaborazione di ogni singolo container che costituisce la vNSF. Nel secondo caso, è evidente come un vNSF Component possa essere aggiornato apportando minime modifiche ai soli file manifest e hostconfig, senza modificare altri dettagli dell'architettura preesistente. Ad esempio, nel caso della vNSF Reverse Proxy, è sufficiente cambiare l'Immagine Docker referenziata nel file manifest per utilizzare una diversa implementazione di Collettore Log.

Inoltre, sebbene si sia registrato un lieve di calo di prestazioni nel collaudo, è indubbio come l'utilizzo di un Reverse Proxy con WAF nel caso Docker sia più performante del caso VM. La condivisione del Sistema Operativo con l'host si dimostra determinante nel garantire un valore di throughput paragonabile a quello del caso Standalone. In aggiunta, il caso Docker non richiede la configurazione manuale di regole con `iptables` e l'installazione dei singoli componenti software come httpd e Logstash. La dipendenza dal solo Docker Daemon rende la vNSF Reverse Proxy maggiormente portabile rispetto alle altre configurazioni.

Infine, è evidente come l'aumento di complessità nell'architettura derivante dall'introduzione di un Load Balancer renda l'ambiente di produzione più performante e tollerante ai guasti. Tale infrastruttura si rivela una scelta obbligata per l'impiego della vNSF Reverse Proxy in ambienti che richiedono un elevato grado di affidabilità.

Bibliografia

[1] P.Mell, T.Grance, "The NIST Definition of Cloud Computing", Special Publication 800-145, September 2011, DOI 10.6028/NIST.SP.800-145

[2] ETSI NFV Industry Specification Group, "Network Functions Virtualisation (NFV); Use Cases ", ETSI GR NFV 001 V1.2.1, Maggio 2017, http://www.etsi.org/deliver/etsi_gr/NFV/001_099/001/01.02.01_60/gr_NFV001v010201p.pdf

[3] VMWare, "Virtualization Essentials", https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/ebook/gated-vmw-ebook-virtualization-essentials.pdf

[4] S.Campbell, M.Jeronimo, "An Introduction to Virtualization", https://software.intel.com/sites/default/files/m/d/4/1/d/8/An_Introduction_to_Virtualization.pdf

[5] Intel Corporation, "Container and Kernel-Based Virtual Machine (KVM) Virtualization for Network Function Virtualization (NFV)", Agosto 2015, https://builders.intel.com/docs/container-and-kvm-virtualization-for-nfv.PDF

[6] W.Felter, A.Ferreira, R.Rajamony, J.Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers", 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia (PA, USA), March 29-31, 2015, pp. 171-172, DOI 10.1109/ISPASS.2015.7095802

[7] A.Grattafiori, "Understanding and Hardening Linux Containers", Giugno 2016, https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf

[8] Docker Documentation, https://docs.docker.com

[9] E.Ekici, "Microservices Architecture, Containers and Docker", Dicembre 2014, https://www.ibm.com/developerworks/community/blogs/1ba56fe3-efad-432f-a1ab-58ba3910b073/entry/microservices_architecture_containers_and_docker?lang=en

[10] R.Rosen, "Linux Containers and the future cloud", Maggio 2013, http://haifux.org/lectures/320/netLec8_final.pdf

[11] R.Pike, D.Presotto, K.Thompson, H.Trickey, P.Winterbottom, "The use of name spaces in plan 9", EW 5 Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring, Mont Saint-Michel (France), September 21-23, 1992, pp. 1-5, DOI 10.1145/506378.506413

[12] The Linux man-pages project, "NAMESPACES - Linux Programmer's Manual", http://man7.org/linux/man-pages/man7/namespaces.7.html

[13] The proc pseudo file-system, http://www.olsr.org/docs/report_html/node75.html

[14] Windows Dev Center, "Interprocess Communications", https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx

[15] M.Kerrisk, "Namespace in operation", Gennaio 2013, https://lwn.net/Articles/531114/

[16] The Linux man-pages project, "PID NAMESPACES - Linux Programmer's Manual", http://man7.org/linux/man-pages/man7/pid_namespaces.7.html

[17] The Linux man-pages project, "USER NAMESPACES - Linux Programmer's Manual", http://man7.org/linux/man-pages/man7/user_namespaces.7.html

[18] CWE-200: Information Exposure, https://cwe.mitre.org/data/definitions/200.html

[19] The Linux man-pages project, "CGROUPS - Linux Programmer's Manual", http://man7.org/linux/man-pages/man7/cgroups.7.html

[20] R.Rosen, "Cgroups V2 overview", Linux Plumbers Conference, Santa Fe (NM, USA), November 1-4, 2016,
[21] T.Heo, "Control Group v2", https://www.kernel.org/doc/Documentation/cgroup-v2.txt
[22] J.H.Saltzer, M.D.Schroeder, "The Protection of Information in Computer System", Proceedings of the IEEE, Vol. 63, No. 9, Settembre 1975, DOI 10.1109/PROC.1975.9939
[23] Testing for Privilege escalation (OTG-AUTHZ-003), https://www.owasp.org/index.php/Testing_for_Privilege_escalation_(OTG-AUTHZ-003)
[24] S.Graber, "LXD 2.0: Introduction to LXD", Novembre 2016, https://stgraber.org/2016/03/11/lxd-2-0-introduction-to-lxd-112/
[25] LXC Introduction, https://linuxcontainers.org/lxc/introduction/
[26] LXD Introduction, https://linuxcontainers.org/lxd/
[27] rkt Overview, https://coreos.com/rkt
[28] LinuxContainers.org, https://linuxcontainers.org/it
[29] D.Lezcano, "LXC container configuration file", https://linuxcontainers.org/it/lxc/manpages/man5/lxc.container.conf.5.html
[30] An OpenStack Compute driver for LXD, https://github.com/openstack/nova-lxd
[31] OpenStack Overview, https://www.openstack.org/software/
[32] Kubernetes Concepts, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[33] Trusted Computing Group, "Trusted Platform Module (TPM) Summary", Aprile 2008, https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/
[34] Trusted Computing Group, "TPM Library Specification", https://trustedcomputinggroup.org/tpm-library-specification/
[35] Docker News and Press, https://www.docker.com/docker-news-and-press/dotcloud-inc-now-docker-inc
[36] DevOps.come and ClusterHQ, "Container Market Adoption", Giugno 2016, https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf
[37] Docker Overview, https://www.docker.com/what-docker
[38] T.Bui, "Analysis of Docker Security", Gennaio 2015,
[39] T.Dierks, E.Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC-5246, August 2008, DOI 10.17487/RFC5246
[40] CIS - Center for Internet Security, https://www.cisecurity.org/
[41] CIS Benchmarks - Securing Docker, https://www.cisecurity.org/benchmark/docker/
[42] P.Goyal, "CIS Docker 1.13.0 Benchmark v1.0.0", Gennaio 2017, https://www.cisecurity.org/benchmark/docker/
[43] shadow(5) - Linux man page, https://linux.die.net/man/5/shadow
[44] J.Halfacre, "Protecting Against Website Defacement", Gennaio 2017, https://www.removemalware.net/website-defacement/
[45] The Linux man-pages project, "UNIX - Linux Programmer's Manual", http://man7.org/linux/man-pages/man7/unix.7.html
[46] J.Hertz, "Abusing Privileged and Unprivileged Linux Containers", Giugno 2016, https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/june/abusing-privileged-and-unprivileged-linux-containers.pdf
[47] Linux Foundation Wiki, "bridge", https://wiki.linuxfoundation.org/networking/bridge
[48] Production Quality, Multilayer Open Virtual Switch, http://openvswitch.org/
[49] D.C.Plummer, "An Ethernet Address Resolution Protocol", RFC-826, November 1982, DOI 10.17487/RFC0826
[50] T.Fox, "Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks", Luglio 2017, https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks
[51] R.Benevides, "Java inside docker: What you must know to not FAIL", Marzo 2017, https://developers.redhat.com/blog/2017/03/14/java-inside-docker/
[52] Introduction to SELinux, https://www.centos.org/docs/5/html/Deployment_Guide-en-US/ch-selinux.html

[53] R.S.Sandhu, P.Samarati, "Access Control: Principles and Practice", IEEE Communications Magazione, Vol. 32, No. 9, Settembre 1994, pp. 40-48, DOI 10.1109/35.312842
[54] D.Downs, J.R.Rub, K.C.Kung, C.S.Jordan, "Issues in Discretionary Access Control", 1985 IEEE Symposium on Security and Privacy, Oakland (CA, USA), April 22-24, 1985, pp. 208-218, DOI 10.1109/SP.1985.10014
[55] P.Loscocco, S.Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, Boston (MA, USA), June 25-30, 2001, pp. 29-42,
[56] S.Smalley, "Configuring the SELinux Policy", Gennaio 2003,
[57] SELinux Project Wiki, https://selinuxproject.org/page/Main_Page
[58] SECure COMPuting with filters, https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
[59] The Linux man-pages project, "SECCOMP - Linux Programmer's Manual", http://man7.org/linux/man-pages/man2/seccomp.2.html
[60] S.McCanne, V.Jacobson, "The BSD Packet Filter: A new Architecture for User-level Packet Capture", USENIX Winter 1993 Conference Proceedings, San Diego (CA, USA), Gennaio 25-29, 1993, pp. 259-270,
[61] J.Edge, "A seccomp overview", Settembre 2015, https://lwn.net/Articles/656307/
[62] Security - Seccomp, https://wiki.tizen.org/Security:Seccomp
[63] T.Combe, A.Martin, R.Di Pietro, "To Docker or Not to Docker: A Security Perspective", IEEE Cloud Computing, Vol. 3, No. 5, Sept-Oct 2016, pp. 54-62 DOI 10.1109/MCC.2016.100
[64] R.Housley, W.Ford, W.Polk, D.Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC-2459, Gennaio 1999, DOI 10.17487/RFC2459
[65] N.Myers, R.Ankney, A.Malpani, S.Galperin, C.Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC-2560, Giugno 1999, DOI 10.17487/RFC2560
[66] Github, https://github.com/
[67] BitBucket, https://bitbucket.org/
[68] I.Cooper, I.Melve, G.Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC-3040, January 2001, DOI 10.17487/RFC3040
[69] A.Stricek, "A Reverse Proxy Is A Proxy By Any Other Name", Gennaio 2002, https://www.sans.org/reading-room/whitepapers/webservers/reverse-proxy-proxy-name-302
[70] Apache HTTP Server Project, https://httpd.apache.org/
[71] June 2017 Web Server Survey, https://news.netcraft.com/archives/2017/06/27/june-2017-web-server-survey.html
[72] R.Fielding, J.Gettys, J.Mogul, H.Frystyk, L.Masinter, P.Leach, T.Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", RFC-2616, June 1999, DOI 10.17487/RFC2616
[73] Traffic Server, http://trafficserver.apache.org/
[74] Apache License Version 2.0, Gennaio 2004, https://www.apache.org/licenses/LICENSE-2.0
[75] Apache Traffic Server - Administrator's Guide, https://docs.trafficserver.apache.org/en/latest/admin-guide/index.en.html
[76] Varnish HTTP Cache, https://varnish-cache.org/
[77] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer, http://www.haproxy.org/
[78] HAProxy Starter Guide, http://cbonte.github.io/haproxy-dconv/1.7/intro.html
[79] HAProxy Management Guide, http://cbonte.github.io/haproxy-dconv/1.7/management.html
[80] Configuring and Using Chroot Jails, https://docs.oracle.com/cd/E37670_01/E36387/html/ol_cj_sec.html
[81] D.Maltz, P.Bhagwat, "TCP Splicing for Application Layer Proxy Performance", Journal of High Speed Networks, Vol. 8, No. 3, pp. 225-240, Settembre 1998,
[82] NGINX Wiki, https://www.nginx.com/resources/wiki/
[83] 2016 Data Breach Investigations Report, http://www.verizonenterprise.com/resources/reports/rp_DBIR_2016_Report_en_xg.pdf

[84] J.Pubal, "Web Application Firewalls", Marzo 2015, https://www.sans.org/reading-room/whitepapers/application/web-application-firewalls-35817
[85] OWASP, https://www.owasp.org/index.php/Main_Page
[86] WASC, http://www.webappsec.org/
[87] WAFEC, https://www.owasp.org/index.php/Projects/WASC_OWASP_Web_Application_Firewall_Evaluation_Criteria_Project/Releases/Current
[88] Web Application Security Consortium, "Web Application Firewall Evaluation Criteria", Gennaio 2006, http://projects.webappsec.org/f/wasc-wafec-v1.0.pdf
[89] OWASP Top Ten Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[90] OWASP, "OWASP Top 10 2013 document", https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/owasptop10/OWASP%20Top%2010%20-%202013.pdf
[91] SQL Injection, https://www.owasp.org/index.php/SQL_Injection
[92] Session Fixation, https://www.owasp.org/index.php/Session_fixation
[93] OWASP Application Security Verification Standard Project, https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project
[94] Cross-site Scripting, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
[95] ModSecurity, https://modsecurity.org/
[96] NAXSI, an Nginx Web Application Firewall, https://www.nbs-system.com/en/it-security/it-security-tools-open-source/naxsi/
[97] Ivan Ristić, "MODSECURITY HANDBOOK", Feisty Duck, 2017, ISBN: 978-1-907117-07-7
[98] L.Saunois, "NAXSI, a web application firewall for Nginx", Giugno 2015, https://www.nbs-system.com/en/blog/naxsi-web-application-firewall-for-nginx/
[99] T.Koechlin, "NAXSI, a web application firewall for Nginx", Settembre 2011, https://www.nbs-system.com/en/blog/naxsi-opensource-application-firewall-for-nginx/
[100] Elasticsearch - The hearth of the Elastic Stack, https://www.elastic.co/products/elasticsearch
[101] Kibana - Your Window into the Elastic Stack, https://www.elastic.co/products/kibana
[102] ab - Apache HTTP server benchmarking tool, https://httpd.apache.org/docs/2.4/programs/ab.html
[103] Apache MPM event, https://httpd.apache.org/docs/2.4/mod/event.html
[104] SHIELD - Securing against intruders and other threats through a NFV-enabled environment, https://www.shield-h2020.eu/
[105] "Specifications, design and architecture for the vNSF ecosystem", Giugno 2017, https://torsec.github.io/shield-h2020/documents/project-deliverables/SHIELD_D3.1_Specifications,_Design_and_Architecture_for_the_vNSF_Ecosystem_v1.0.pdf
[106] ETSI NFV Industry Specification Group, "Network Functions Virtualisation (NFV); Virtual Network Functions Architecture ", ETSI GS NFV-SWA 001 V1.1.1, Dicembre 2014, http://www.etsi.org/deliver/etsi_gs/NFV-SWA/001_099/001/01.01.01_60/gs_NFV-SWA001v010101p.pdf
[107] The SECURED project, http://www.secured-fp7.eu/
[108] D4.1 Policy specification, SECURED Project deliverables, Marzo 2015, http://www.secured-fp7.eu/files/secured_d41_policy_spec_v0100.pdf
[109] Docker Engine API (v1.32), https://docs.docker.com/engine/api/v1.32/
[110] Ceph Documentation, http://docs.ceph.com/docs/jewel/
[111] HDFS Architecture Guide, https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[112] M.Mahalingam, D.Dutt, K.Duda, P.Agarwal, L.Kreeger, T.Sridhar, M. Bursell, C.Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC-7348, August 2014, DOI 10.17487/RFC7348
[113] ETSI NFV Industry Specification Group "Network Functions Virtualisation (NFV); Architectural Framework ", ETSI GS NFV 002 V1.1.1, Ottobre 2013, http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf
[114] etcd, https://coreos.com/etcd
[115] Consul, https://www.consul.io/
[116] Apache ZooKeeper, https://zookeeper.apache.org/
[117] The Go Programming Language, https://golang.org/

[118] Docker Engine SDK reference, https://godoc.org/github.com/moby/moby/client
[119] ModSecurity - Reference Manual, https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual
[120] Logstash, https://www.elastic.co/products/logstash
[121] Grok Filter Plugin, https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html
[122] logstash-modsecurity, https://github.com/bitsofinfo/logstash-modsecurity
[123] The netfilter.org project, http://www.netfilter.org/

Appendice A

Manuale Utente

Le procedure esaminate in questo Manuale Utente consentono di installare e configurare il software necessario per il collaudo della vNSF Reverse Proxy. Se non diversamente specificato, ogni sezione prevede l'utilizzo di una distribuzione CentOS 7 con kernel Linux 3.10.0 (https://www.centos.org/download/).

A.1 Installazione Docker in CentOS 7

Prerequisiti: nessuno

La piattaforma Docker è disponibile in due versioni: *Community Edition* (CE) ed *Enterprise Edition* (EE). La prima è maggiormente orientata allo sviluppo, mentre la seconda è più idonea ad ambienti di produzione. La procedura qui indicata consente di installare Docker nella versione CE. I vari passaggi possono essere approfonditi consultando la documentazione ufficiale all'indirizzo https://docs.docker.com/engine/installation/linux/docker-ce/centos. Si ricorda che è necessario che l'utente utilizzato per seguire la procedura sia dotato dei privilegi di amministratore sul Sistema Operativo.

Prima di procedere al download di Docker è necessario installare preliminarmente alcuni pacchetti con il gestore `yum`:

```
[shield@localhost ~]$ sudo yum -y install yum-utils \
    device-mapper-persistent-data lvm2
```

Successivamente, procedere all'aggiunta nel gestore `yum` del repository `stable`, contenente una versione stabile di Docker CE:

```
[shield@localhost ~]$ sudo yum-config-manager --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

Una volta abilitato il repository, si può passare allo scaricamento e all'installazione di Docker:

```
[shield@localhost ~]$ sudo yum -y install docker-ce
```

Essendo il primo pacchetto scaricato dal repository, sarà richiesto di accettare la chiave GPG. Prima di accettarla, verificarla che sia equivalente a quella riportata all'indirizzo https://docs.docker.com/engine/installation/linux/docker-ce/centos.

Una volta completata l'installazione, si può avviare il servizio corrispondente all'interno di CentOS 7:

```
[shield@localhost ~]$ sudo systemctl start docker
```

Per verificare l'esito dell'installazione è possibile testare il funzionamento con il comando:

```
[shield@localhost ~]$ sudo docker run hello-world
```

Se ricevuto un messaggio di benvenuto, l'installazione è andata a buon fine. In caso contrario, consultare la documentazione ufficiale.

A.2 Installazione e configurazione di un cluster etcd

Prerequisiti: Installazione Docker in CentOS 7

Il cluster etcd utilizzato nel collaudo della vNSF Reverse Proxy conta tre differenti VM, appartenenti alla sottorete `192.168.0.0/24`:

- hostname: `etcd-node-1`, indirizzo IP: `192.168.0.101`
- hostname: `etcd-node-2`, indirizzo IP: `192.168.0.102`
- hostname: `etcd-node-3`, indirizzo IP: `192.168.0.103`

Ciascuna di esse può essere contattata per recuperare i valori salvati all'interno del Key/Value Storage.

Prima di procedere alla fase di installazione è necessario installare il software `wget` per procurarsi l'eseguibile di `etcd`. Tale operazione è da effettuare su ciascuno dei tre nodi:

```
[shield@localhost ~]$ sudo yum -y install wget
```

Completata questa fase preliminare, procedere al download dalla pagina https://github.com/coreos/etcd/releases/ dell'ultima versione precompilata (attualmente 3.2.9):

```
[shield@localhost ~]$ sudo wget \
github.com/coreos/etcd/releases/download/v3.2.9/etcd-v3.2.9-linux-amd64.tar.gz
```

Successivamente, decomprimere l'archivio con il comando `tar`:

```
[shield@localhost ~]$ tar xvf etcd-v3.2.9-linux-amd64.tar.gz
```

Infine, aggiungere la directory appena scaricata alla variabile `PATH`:

```
[shield@localhost ~]$ cd etcd-v3.2.9-linux-amd64
[shield@localhost etcd-v3.2.9-linux-amd64]$ PATH = $PATH:$PWD
```

Alternativamente, è possibile clonare, con il software `git`, il codice sorgente dal repository https://github.com/coreos/etcd e compilarlo manualmente.

A.2.1 Struttura etcd

Prima di procedere alla configurazione di ogni host, è opportuno premettere che all'interno di `etcd` si distinguono le figure di `peer` e `client`. I primi costituiscono il Key/Value Storage distribuito e custodiscono i dati, mentre i secondi si occupano del salvataggio e del recupero delle coppie chiave-valore.

`etcd` prevede di default l'utilizzo di due porte: la porta 2379 per la comunicazione tra `client` e `peer` e la porta 2380 per quella tra `peer`.

A.2.2 Configurazione dei peer

Nell'introduzione a questa sezione è stato anticipato che il cluster `etcd` utilizzato conta tre diverse VM. Ognuna di esse rappresenta un `peer` all'interno della configurazione di `etcd`.

I flag utilizzati nella configurazione sono:

`--name` nome di dominio;

`--initial-advertise-peer-urls` durante il bootstrap, specifica una lista di indirizzi IP o nomi di dominio e porte tramite i quali il `peer` è raggiungibile dagli altri `peer`. Almeno uno tra gli URL elencati deve poter essere contattato da tutti gli altri `peer`;

`--initial-cluster-token` durante il bootstrap, specifica un token per il cluster che si sta configurando es. `etcd-cluster`;

`--initial-cluster` durante il bootstrap, indica la configurazione iniziale del cluster. Il suo contenuto deve essere coerente con quanto specificato nel flag `--initial-advertise-peer-urls` di ogni `peer`;

`--advertise-client-urls` specifica una lista di indirizzi IP o nomi di dominio tramite i quali il `peer` è raggiungibile dai `client`;

`--listen-peer-urls` specifica una lista di indirizzi IP e porte sui quali un `peer` è in ascolto per il traffico con i `peer`. Non è possibile utilizzare nomi di dominio;

`--listen-client-urls` specifica una lista di indirizzi IP e porte sui quali un `peer` è in ascolto per il traffico con i `client`. Non è possibile utilizzare nomi di dominio;

`--peer-cert-file` nella comunicazione tra `peer`, specifica il file contenente il certificato del `peer` per l'utilizzo del protocollo TLS;

`--peer-key-file` nella comunicazione tra `peer`, specifica il file contenente la chiave privata del `peer` per l'utilizzo del protocollo TLS;

`--peer-trusted-ca-file` nella comunicazione tra `peer`, specifica il certificato della CA ritenuta fidata per l'utilizzo di TLS;

`--peer-client-cert-auth` abilita l'autenticazione dei client mediante certificato;

`--key-file` nella comunicazione tra `client` e `peer`, specifica il file contenente la chiave privata del `peer` per l'utilizzo del protocollo TLS;

`--cert-file` nella comunicazione tra `client` e `peer`, specifica il file contenente il certificato del `peer` per l'utilizzo del protocollo TLS;

`--trusted-ca-file` nella comunicazione tra `client` e `peer`, specifica il certificato della CA ritenuta fidata per l'utilizzo di TLS.

I flag aventi il prefisso `--initial` sono considerati solo al primo avvio del cluster e sono ignorati negli avvii successivi. Per approfondire, è opportuno consultare la documentazione ufficiale, presente all'indirizzo https://github.com/coreos/etcd/blob/master/Documentation/v2/README.md.

Configurazione `etcd-node-1`

Per avviare `etcd` sul nodo `etcd-node-1`, utilizzare il comando:

```
[shield@etcd-node-1 ~]$ etcd --name etcd-node-1 \
    --initial-advertise-peer-urls https://192.168.0.101:2380 \
    --initial-cluster-token etcd-cluster \
    --initial-cluster etcd-node-1=https://192.168.0.101:2380,\
        etcd-node-2=https://192.168.0.102:2380,\
        etcd-node-3=https://192.168.0.103:2380
    --advertise-client-urls https://192.168.0.101:2379 \
    --listen-peer-urls https://192.168.0.101:2380 \
    --listen-client-urls https://192.168.0.101:2379 \
    --peer-cert-file certs/etcd-node-1.pem \
    --peer-key-file certs/etcd-node-1_key.pem \
    --peer-trusted-ca-file certs/ca.pem
    --peer-client-cert-auth
    --cert-file certs/etcd-node-1.pem
    --key-file certs/etcd-node-1_key.pem
    --trusted-ca-file certs/ca.pem
```

Configurazione `etcd-node-2`

Per avviare `etcd` sul nodo `etcd-node-2`, utilizzare il comando:

```
[shield@etcd-node-2 ~]$ etcd --name etcd-node-2 \
    --initial-advertise-peer-urls https://192.168.0.102:2380 \
    --initial-cluster-token etcd-cluster \
    --initial-cluster etcd-node-1=https://192.168.0.101:2380,\
        etcd-node-2=https://192.168.0.102:2380,\
        etcd-node-3=https://192.168.0.103:2380
    --advertise-client-urls https://192.168.0.102:2379 \
    --listen-peer-urls https://192.168.0.102:2380 \
    --listen-client-urls https://192.168.0.102:2379 \
    --peer-cert-file certs/etcd-node-2.pem \
    --peer-key-file certs/etcd-node-2_key.pem \
    --peer-trusted-ca-file certs/ca.pem
    --peer-client-cert-auth
    --cert-file certs/etcd-node-2.pem
    --key-file certs/etcd-node-2_key.pem
    --trusted-ca-file certs/ca.pem
```

Configurazione `etcd-node-3`

Per avviare `etcd` sul nodo `etcd-node-3`, utilizzare il comando:

```
[shield@etcd-node-3 ~]$ etcd --name etcd-node-3 \
    --initial-advertise-peer-urls https://192.168.0.103:2380 \
    --initial-cluster-token etcd-cluster \
    --initial-cluster etcd-node-1=https://192.168.0.101:2380,\
        etcd-node-2=https://192.168.0.102:2380,\
        etcd-node-3=https://192.168.0.103:2380
    --advertise-client-urls https://192.168.0.103:2379 \
    --listen-peer-urls https://192.168.0.103:2380 \
    --listen-client-urls https://192.168.0.103:2379 \
    --peer-cert-file certs/etcd-node-3.pem \
    --peer-key-file certs/etcd-node-3_key.pem \
    --peer-trusted-ca-file certs/ca.pem
    --peer-client-cert-auth
    --cert-file certs/etcd-node-3.pem
```

```
--key-file certs/etcd-node-3_key.pem
--trusted-ca-file certs/ca.pem
```

Verifica della configurazione

Per verificare il funzionamento del cluster appena configurato è possibile utilizzare lo strumento `etcdctl`, fornito congiuntamente all'eseguibile `etcd`. Nello specifico, si utilizzano i seguenti flag:

`--ca-file` specifica il certificato della CA fidata;

`--key-file` specifica la chiave privata del client;

`--cert-file` specifica il certificato da utilizzare per l'autenticazione presso i `peer`;

`--endpoints` lista degli indirizzi IP dei `peer` all'interno del cluster.

Il comando diviene pertanto:

```
[shield@localhost ~]$ etcdctl --ca-file certs/ca.pem \
   --key-file certs/shield-vnsf_1-key.pem \
   --cert-file certs/shield-vnsf_1.pem \
   --endpoints https://192.168.0.101:2379,\
      https://192.168.0.102:2379,\
      https://192.168.0.103:2379\
   cluster-health
```

Nel caso in cui non vi siano errori, l'output del comando segnala la corretta configurazione del cluster:

```
member aeed280fbd4b48b is healthy: got healthy result from \
https://192.168.0.101:2379
member 1e149427bfba9593 is healthy: got healthy result from \
https://192.168.0.102:2379
member 4623932738f9a61d is healthy: got healthy result from \
https://192.168.0.103:2379
cluster is healthy
```

A.2.3 Configurazione dei client

Per consentire l'utilizzo del driver di rete `overlay` in Docker è necessario che il Docker Daemon possa interfacciarsi con il Key/Value Storage, `etcd` nel caso di studio. All'interno di un cluster `etcd`, il Docker Daemon svolge il ruolo di `client`.

In ogni VM equipaggiata con la piattaforma Docker, all'interno della sottorete `192.168.0.0/24`, aprire, mediante un editor di testo (es. `vi`), il file di configurazione del servizio Docker:

```
[shield@localhost ~]$ sudo vi /usr/lib/systemd/system/docker.service
```

Cercare la direttiva `ExecStart=/usr/bin/dockerd` e aggiungere i flag:

`--cluster-store` indirizzo dei `peer` nel cluster `etcd`.

`--cluster-advertise` interfaccia e porta esposta nel `client`.

`--cluster-store-opt` opzioni per l'interfacciamento con il cluster.

Considerando la configurazione di rete utilizzata finora, il risultato è il seguente:

```
ExecStart=/usr/bin/dockerd --cluster-store=etcd://192.168.0.101:2379,\
     192.168.0.102:2379,\
     192.168.0.103:2379 \
    --cluster-advertise=eth0:2379 \
    --cluster-store-opt kv.cacertfile=/home/shield/certs/ca.pem \
    --cluster-store-opt kv.keyfile=/home/shield/certs/shield-key.pem \
    --cluster-store-opt kv.certfile=/home/shield/certs/shield.pem
```

Per poter effettuare le modifiche, è necessario riavviare il Docker Daemon eseguendo nell'ordine:

```
[shield@localhost ~]$ sudo systemctl daemon-reload
```

e poi:

```
[shield@localhost ~]$ sudo systemctl restart docker.service
```

Al fine di verificare il funzionamento del cluster `etcd` è possibile creare una rete `sample` con il driver di rete `overlay`:

```
[shield@localhost ~]$ sudo docker network create -d overlay sample
```

In caso di esito positivo, è possibile ritrovare tale rete all'interno della lista fornita dal comando:

```
[shield@localhost ~]$ sudo docker network ls | grep sample
0d1b7c5899ea    sample    overlay    global
```

A.3 Abilitazione supporto SELinux per Docker

Prerequisiti: Installazione Docker in CentOS 7

Per poter abilitare il supporto SELinux in Docker è necessario modificare la configurazione del Docker Daemon secondo la seguente procedura. Si ricorda che è necessario che l'utente utilizzato per seguire la procedura sia dotato dei privilegi di amministratore sul Sistema Operativo.

Aprire, mediante un editor di testo (es. `vi`), il file di configurazione del servizio Docker:

```
[shield@localhost ~]$ sudo vi /usr/lib/systemd/system/docker.service
```

Cercare la direttiva `ExecStart=/usr/bin/dockerd` e aggiungere in coda la stringa `--selinux-enabled` come riportato di seguito:

```
ExecStart=/usr/bin/dockerd --selinux-enabled
```

Per poter effettuare le modifiche, è necessario riavviare il Docker Daemon eseguendo nell'ordine:

```
[shield@localhost ~]$ sudo systemctl daemon-reload
```

e poi:

```
[shield@localhost ~]$ sudo systemctl restart docker.service
```

A.3.1 Configurazione auditd

Opzionalmente, si può modificare la configurazione di `auditd` di CentOS per indicare quali eventi bloccati da SELinux riportare nei file di log. La procedura che segue consente di riportare nei file di log i tentativi di accesso a directory bloccati.

Aprire, con un editor di testo, il file `/etc/audit/rules.d/audit.rules` e aggiungere in coda le seguenti direttive:

```
-a always,exit -F arch=b64 -S open,openat,chdir -F exit=-EACCES -k access
-a always,exit -F arch=b64 -S open,openat,chdir -F exit=-EPERM -k access
-a always,exit -F arch=b32 -S open,openat,chdir -F exit=-EACCES -k access
-a always,exit -F arch=b32 -S open,openat,chdir -F exit=-EPERM -k access
```

Queste indicano di riportare le situazioni in cui le syscall `open`, `openat` e `chdir` per l'accesso ad un file o ad una directory falliscono per mancanza dei relativi permessi.

Per rendere effettive le modifiche, riavviare il demone `auditd`:

```
[shield@localhost ~]$ sudo service auditd restart
```

A.3.2 Esempio di Type Enforcement

Per evitare la modifica accidentale di configurazioni preesistenti, creare una cartella di test all'interno della directory home dell'utente:

```
[shield@localhost ~]$ cd
[shield@localhost ~]$ mkdir sandbox
[shield@localhost ~]$ cd sandbox
[shield@localhost sandbox]$
```

A partire dalla cartella `sandbox`, creare una nuova cartella e si verifichi il contesto di sicurezza nel seguente modo:

```
[shield@localhost sandbox]$ mkdir example
[shield@localhost sandbox]$ ls -Z
drwxr-xr-x. shield shield unconfined_u:object_r:user_home_t:s0 example
```

Si nota che il tipo assegnato da SELinux alla cartella appena creata è `user_home_t`. Avviare ora un container all'interno della directory `sandbox` mediante la seguente direttiva:

```
[shield@localhost ~]$ sudo docker run -ti --rm -v $HOME/sandbox:/opt \
   centos:7 /bin/bash
```

la quale consente di avviare un container a partire dall'Immagine `centos:7`, creare un binding tra la directory di lavoro (`$HOME/sandbox`) e il percorso `/opt` all'interno del container ed eseguire al suo interno una shell `/bin/bash`. In caso di successo, il comando appena digitato restituisce un prompt di questo tipo:

```
[root@5a5e6b63cb29 /]#
```

per l'esecuzione di comandi all'interno del container. A questo punto, il tentativo di elencare il contenuto della directory `/opt` ha il seguente risultato:

```
[root@5a5e6b63cb29 /]# ls /opt
ls: cannot open directory /opt: Permission denied
```

Il motivo di tale risultato è giustificato dall'utilizzo del Type Enforcement. I processi in esecuzione all'interno di un container possiedono la label `svirt_lxc_net_t`, mediante la quale non è possibile accedere ai file etichettati con la label `user_home_t` citata prima.

Nel caso in cui sia stato configurato il demone `auditd` come riportato nell'Appendice A.3.1, è possibile controllare il contenuto del log di SELinux:

```
[shield@localhost ~]$ sudo ausearch --message SYSCALL --success no \
--interpret
...
type=SYSCALL msg=audit(30/10/2017 22:37:12.498:8485) : arch=x86_64 \
syscall=openat success=no exit=EACCES(Permesso negato) \
a0=0xffffffffffffff9c a1=0x25715b0 \
```

```
a2=O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC a3=0x0 items=1 ppid=7247 \
pid=7544 auid=unset uid=root gid=root euid=root suid=root fsuid=root \
egid=root sgid=root fsgid=root tty=pts0 ses=unset comm=ls exe=/usr/bin/ls \
subj=system_u:system_r:svirt_lxc_net_t:s0:c116,c367 key=access
```

Tra le varie voci si comprende che la `syscall=openat`, invocata dall'utente `root` all'interno del container tramite il comando `exe=/usr/bin/ls`, è stata negata (`success=no`).

Uscire dal container e avviare ora un nuovo container simile al precedente, ma con l'indicazione `:z`. Essa indica al Docker Daemon di re-impostare il tipo assegnato al contenuto della directory `sandbox`:

```
[root@5a5e6b63cb29 /]# exit
[shield@localhost ~]$ sudo docker run -ti --rm -v $HOME/sandbox:/opt:z \
    centos:7 /bin/bash
[root@48148d7a2ef6]#
```

In questo caso, il tentativo di leggere la directory avrà successo e potrà essere visionata la cartella creata in precedenza:

```
[root@48148d7a2ef6]# cd /opt
example
[root@48148d7a2ef6]#
```

Uscire dal container e controllare come l'etichetta di tipo di SELinux per la cartella `sandbox` sia stata modificata in `svirt_sandbox_file_t`:

```
[root@48148d7a2ef6]# exit
[shield@localhost ~]$ ls -Z
drwxrwxr-x. shield shield system_u:object_r:svirt_sandbox_file_t:s0 sandbox
```

L'esempio appena riportato dimostra come il supporto SELinux impedisca ad un container compromesso di accedere liberamente al filesystem dell'host, perché etichettati diversamente.

Si ricorda che il Type Enforcement non ha alcun effetto nel caso in cui un container tenti di accedere a file appartenenti ad un altro container, poiché entrambi presentano la stessa etichetta di tipo. In tal caso è necessario adoperare un modello di Multi Category Support.

A.3.3 Esempio di Multi Category Support

Se non fatto precedentemente, creare una directory `sandbox` per evitare modifiche accidentali della configurazione preesistente:

```
[shield@localhost ~]$ cd
[shield@localhost ~]$ mkdir sandbox
[shield@localhost ~]$ cd sandbox
[shield@localhost sandbox]$
```

Questo esempio richiede l'utilizzo di due container differenti, pertanto è consigliato l'utilizzo di due terminali. Creare una cartella all'interno della directory `sandbox` e controllarne il contesto di sicurezza:

```
[shield@localhost ~]$ cd sandbox
[shield@localhost sandbox]$ mkdir example2
[shield@localhost sandbox]$ ls -Z
drwxr-xr-x. shield shield unconfined_u:object_r:user_home_t:s0 example2
```

Avviare ora primo container mediante la direttiva:

```
[shield@localhost ~]$ sudo docker run -ti --rm --name container1 -v
$HOME/sandbox:/opt:z centos:7 /bin/bash
[root@0be9820f0816 /]#
```

Questo comporta l'avvio di un container con le seguenti caratteristiche:

- Immagine `centos:7` di partenza;
- binding tra la directory `sandbox` e il path `/opt` all'interno del container;
- nome `container1`;
- direttiva `:z` per il solo Type Enforcement.

In un secondo terminale, si avvii un altro container (`container2`) senza specificare la sequenza `:z`, con la seguente direttiva:

```
[shield@localhost ~]$ sudo docker run -ti --rm --name container2 \
   -v $HOME/sandbox:/opt centos:7 /bin/bash
[root@48bfc38cd8cd /]#
```

Si noti come, a differenza dell'esempio precedente, il secondo container possa liberamente accedere al contenuto della cartella `sandbox`:

```
[root@48bfc38cd8cd /]# cd opt
example2
```

Questo è dovuto al fatto che entrambi i container hanno gli stessi permessi di operare sul tipo `svirt_sandbox_file_t`.

Uscire da entrambi i container con il comando `exit`. Nel primo terminale, avviare un container con la direttiva:

```
[shield@localhost ~]$ sudo docker run -ti --rm --name container3 \
  -v $HOME/sandbox:/opt:z \
  --security-opt="label:level:s0:c3" \
  centos:7 /bin/bash
[root@bcbb8e4aead6 /]#
```

la quale segnala di re-impostare l'etichetta della directory `sandbox`, già di tipo `svirt_sandbox_file_t`, aggiungendo l'indicazione di sensibilità `s0` e categoria `c3`.

Infine, avviare nel secondo terminale un quarto container con la stessa direttiva utilizzata precedentemente:

```
[shield@localhost ~]$ sudo docker run -ti --rm --name container4 \
-v $HOME/sandbox:/opt centos:7 /bin/bash
[root@2affa77f5c04 /]#
```

In questo caso, un tentativo di accedere alla directory `/opt` all'interno del container si rivelerà fallimentare:

```
[root@2affa77f5c04 /]# ls opt
ls: cannot open directory opt: Permission denied
[root@2affa77f5c04 /]#
```

Nel caso in cui sia stato configurato il demone `auditd` come riportato nell'Appendice A.3.1, è possibile controllare il contenuto del log di SELinux:

```
[shield@localhost ~]$ sudo ausearch --message SYSCALL --success no \
--interpret
...
type=SYSCALL msg=audit(02/11/2017 00:34:16.697:8774) : arch=x86_64 \
syscall=openat success=no exit=EACCES(Permesso negato) a0=0xffffffffffffff9c \
a1=0x16005b0 a2=O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC a3=0x0 items=1 \
ppid=8255 pid=8313 auid=unset uid=root gid=root euid=root suid=root \
fsuid=root egid=root sgid=root fsgid=root tty=pts0 ses=unset comm=ls \
exe=/usr/bin/ls subj=system_u:system_r:svirt_lxc_net_t:s0:c243,c548 key=access
```

Da notare come le categorie `c243` e `c548` riportate nell'ultimo rigo, assegnate arbitrariamente da SELinux per `container4`, non consentano neanche ad un processo etichettato con il tipo `svirt_lxc_net_t` di accedere alla directory. Al fine di concedere la condivisione di file tra `container3` e `container4` è necessario utilizzare l'opzione `--security-opt` all'avvio del `container4` e specificare la stessa sensibilità e categoria specificati per il `container3`.

A.4 Utilizzo di vNSF Controller

Prerequisiti: Installazione Docker in CentOS 7

A.4.1 Download del vNSF Controller

vNSF Controller può essere scaricato direttamente dal relativo repository. Se non fatto in precedenza, installare all'interno del sistema il software `git`:

```
[shield@localhost ~]$ sudo yum -y install git
```

Completata l'installazione, si può procedere alla clonazione del repository `https://github.com/vins1993/shield-thesis` nel seguente modo:

```
[shield@localhost ~]$ git clone https://github.com/vins1993/shield-thesis.git
```

All'interno della directory `shield-thesis/vNSF_Controller` è possibile ritrovare una versione precompilata del vNSF Controller. Se desiderato, qualora vi fosse installato il linguaggio Go nel sistema, è possibile procedere alla ricompilazione del codice in questo modo:

```
[shield@localhost ~]$ cd shield-thesis/vNSF_Controller
[shield@localhost shield-thesis/vNSF_Controller]$ go build
```

Al termine della ricompilazione sarà disponibile un file eseguibile denominato `vNSF_Controller`.

A.4.2 Utilizzo del vNSF Controller

Il vNSF Controller può essere utilizzato per tre funzioni: avvio, lettura dello stato e arresto di una vNSF. La sintassi per invocarlo è la seguente:

```
vNSF_Controller [OPTIONS] arg
```

Il ruolo del campo `arg` dipende dalla funzione richiesta. Nel caso di avvio di una vNSF, `arg` specifica un archivio contenente i file necessari per l'avvio di una vNSF, come manifest e configurazione MSPL. Nei casi di lettura dello stato e arresto di una vNSF, esso specifica il file *vNSF_ID* _status.json contenente la descrizione della vNSF in esecuzione.

Di seguito è disponibile la descrizione della lista di `[OPTIONS]`:

`--api` specifica la versione della Docker API con la quale interfacciarsi con il Docker Daemon. Di default, è impostata la versione `1.32`;

`--deletevolume` può essere utilizzato solo congiuntamente all'opzione `--stop`. Consente di rimuovere i volumi associati ai container facenti parte della vNSF. Di default, tale opzione è disabilitata per consentire analisi successive sul contenuto dei volumi;

`--endpoint` indica il socket su cui è in ascolto il Docker Daemon. Quest'ultimo può essere in ascolto su un socket TCP, UNIX o allo stesso tempo su entrambi i tipi. Di default, il Docker Daemon è contattato sul socket UNIX `unix:///var/run/docker.sock`;

`--hostconfig` specifica il file hostconfig della vNSF per cui si chiede l'avvio. Tale file è ricercato all'interno dell'archivio passato a vNSF Controller come argomento. Di default, il software ricerca un file denominato `hostconfig.conf`;

`--lfile` specifica un file per i log di vNSF Controller. Se non specificato, si assume come file di output `stdout`;

`--lformat` specifica il formato dei log di vNSF Controller. Questi possono essere sia in formato `text` sia `json`. Di default, si utilizza il formato `text`;

`--manifest` specifica il file manifest della vNSF per cui si chiede l'avvio. Tale file è ricercato all'interno dell'archivio passato a vNSF Controller come argomento. Di default, il software ricerca un file denominato `manifest.conf`;

`--mode` specifica il livello di dettaglio dell'output del vNSF Controller. Vi sono cinque diversi livelli di dettaglio:

`debug` estremamente verboso, utile per gli sviluppatori;

`info` informazioni generali sullo stato del vNSF Controller;

`warning` avvisi non critici per l'esecuzione del programma;

`error` errori non critici per l'esecuzione del programma;

`fatal` errori gravi per cui è stata richiamata la funzione `exit` es. file non trovato;

`panic` situazione di *panic* all'interno del programma es. errori di programmazione.

Consultare la documentazione ufficiale del progetto https://github.com/sirupsen/logrus per ulteriori informazioni. Di default, il livello di dettaglio utilizzato è `info`;

`--mspl` specifica la configurazione MSPL della vNSF per cui si chiede l'avvio. Tale file è ricercato all'interno dell'archivio passato a vNSF Controller come argomento. Di default, il software ricerca un file denominato `mspl.xml`;

`--status` impostato su `true`, restituisce lo stato della vNSF il cui file *vNSF_ID* _status.json è passato come argomento. Di default, il flag ha il valore `false`;

`--stop` impostato su `true`, arresta la vNSF il cui file *vNSF_ID* _status.json è passato come argomento. Di default, il flag ha il valore `false`;

`--store` specifica il Docker Registry dal quale scaricare le immagini Docker nelle operazioni di Build o di Pull. Se non definito, vNSF Controller provvede al download di tutti i componenti richiesti dal Docker Hub.

A.4.3 Esempio di utilizzo

Si consideri una vNSF, identificata con `abcdef1234356`, con configurazione in MSPL nel file `conf.xml`, file manifest denominato `conf.json` e file hostconfig denominato `hostconfig.conf`. Questi sono inseriti in un archivio denominato `vnsf.tar.gz`. Si vuole procedere all'avvio della vNSF contattando il Docker Daemon in ascolto su un socket TCP, utilizzando la versione `1.30` della Docker API. Il comando da utilizzare è il seguente:

```
vNSF_Controller --api 1.30 \
    --endpoint tcp://192.168.0.2:2375 \
    --manifest conf.json
    --mspl conf.xml \
    vnsf.tar.gz
```

Al termine del processo, vNSF Controller produce il file `abcdef123456_status.json`. A partire da questo è possibile leggere lo stato della vNSF eseguendo il comando:

```
vNSF_Controller --status=true abcdef123456_status.json
```

Per arrestare la vNSF vi sono due comandi diversi, a seconda che si vogliano preservare o meno i volumi associati. Nel primo caso, utilizzare:

```
vNSF_Controller --stop=true abcdef123456_status.json
```

mentre nel secondo:

```
vNSF_Controller --stop=true --deletevolume=true abcdef123456_status.json
```

A.4.4 Avvio dei test predefiniti

È possibile testare il funzionamento del vNSF Controller mediante alcuni test già predefiniti. Questi ultimi sono disponibili all'interno del repository già citato nell'Appendice A.4.1, all'interno della directory `shield-thesis/tests`. Per avviarli, è sufficiente utilizzare lo script Python `vNSF_tester.py`:

```
[shield@localhost shield-thesis/vNSF_Controller]$ sudo python vNSF_tester.py
Syntax: vNSF_tester.py -t <testcase>
```

Sono stati configurati i seguenti scenari di test:

test1 utilizzo della vNSF Reverse Proxy con un Origin Server in esecuzione sullo stesso host. Il WAF è abilitato;

test2 utilizzo della vNSF Reverse Proxy con un Origin Server in esecuzione su host remoto. Il WAF è abilitato;

test2_disabled utilizzo della vNSF Reverse Proxy con un Origin Server in esecuzione su host remoto. Il WAF è disabilitato;

test3 utilizzo di più istanze della vNSF Reverse Proxy con un Origin Server in esecuzione su host remoto. Il WAF è abilitato.

Ad eccezione del caso **test1**, è necessario l'utilizzo di più host (o più VM) per ciascuno degli scenari succitati. Inoltre, sempre ad eccezione del caso **test1**, è necessario modificare i tag `<destination-address>`, `<destination-port>` e `<destination-path>` dei file `mspl.xml` opportunamente con i dati dell'host remoto.

Pertanto, per testare il funzionamento della vNSF Reverse Proxy localmente, si può procedere con il seguente comando:

```
[shield@localhost shield-thesis/vNSF_Controller]$ sudo python vNSF_tester.py \
   -t test1
```

Al termine dell'esecuzione è possibile verificare l'avvio della vNSF controllando l'ID, lo stato e l'Immagine di partenza dei container in esecuzione sull'host:

```
[shield@localhost shield-thesis/vNSF_Controller]$ docker container ls \
   --format="{{.ID}} {{.Status}} {{.Image}}" --all
fd859a11f68f Up 5 minutes (healthy) collector_image
45fc552049f1 Up 5 minutes (healthy) reverseproxy_image
7c50899e4418 Up 5 minutes           sampleserver
882e0b861658 Exited (0) 5 minutes ago mspltranslator_image
```

A.5 Installazione di `ab`

Prerequisiti: nessuno

Il software di benchmark `ab` può essere utilizzato per testare le prestazioni della vNSF Reverse Proxy. Per procedere al download è sufficiente eseguire il seguente comando:

```
[shield@localhost ~]$ sudo yum -y install httpd-tools
```

Terminata l'installazione, è possibile utilizzarlo seguendo la sintassi:

```
ab [OPTIONS] URL
```

Le opzioni utilizzate nel corso della trattazione sono le seguenti:

`-c` imposta il livello di concorrenza simulando client differenti;

`-k` imposta l'intestazione `Connection: keep-alive` per le richieste HTTP. Pertanto, mantiene un'unica sessione HTTP per client;

`-n` imposta il numero di richieste da effettuare nei confronti dell'URL specificato;

`-t` imposta un limite alla durata del test.

Ad esempio per simulare un flusso di 50000 richieste HTTP/1.0 per la pagina `www.example.org/index.html` è possibile eseguire il seguente comando:

```
ab -n 50000 http://www.example.org/index.html
```

Ovvero, per creare un flusso di richieste della durata di 60 s utilizzando connessioni persistenti, simulando una concorrenza di 50 utenti, è possibile eseguire:

```
ab -k -t60s -c 50 http://www.example.org/index.html
```

A.6 Test automatici con vNSF_Client_tester.py

Prerequisiti: Installazione di `ab`

È possibile effettuare automaticamente il test di una vNSF Reverse Proxy in esecuzione mediante lo script Python `vNSF_Client_tester.py`, presente all'interno del progetto Github https://github.com/vins1993/shield-thesis.git.

Se non fatto in precedenza, installare all'interno del sistema il software `git`:

```
[shield@localhost ~]$ sudo yum -y install git
```

Completata l'installazione, si può procedere alla clonazione del repository https://github.com/vins1993/shield-thesis nel seguente modo:

```
[shield@localhost ~]$ git clone https://github.com/vins1993/shield-thesis.git
```

Lo script `vNSF_Client_tester.py` è presente nella directory `shield-thesis/tests`. La sintassi per l'esecuzione è la seguente:

```
python vNSF_Client_tester.py [OPTIONS] URL
```

L'argomento `URL` indica la risorsa richiesta. Le opzioni attualmente disponibili sono:

`-c` specifica il numero di client concorrenti simulati per l'esecuzione di ogni iterazione del test. Di default, è impostata una concorrenza pari a 50 client;

`-d` specifica la directory in cui saranno salvati i file di output dello script. Di default, è utilizzata la directory `output`;

`-h` mostra le opzioni disponibili per l'esecuzione dello script;

`-i` specifica il numero di iterazioni consecutive per l'esecuzione del test. Di default, è impostato un numero di iterazioni pari a 3;

`-k` impone l'utilizzo dell'intestazione `Connection: keep-alive`;

`-n` specifica il numero di richieste da effettuare per ogni iterazione del test. Se non specificato, in presenza dell'impostazione `-t`, esso diviene pari a 50000. Si veda la documentazione di `ab`, mediante il comando `man ab`, per approfondire. Di default, è impostato un numero di richieste pari a 1000 per ogni iterazione;

`-o` specifica il file CSV, all'interno della directory indicata da `-d`, in cui sono riportati i valori di throughput, in KiB/s, ottenuti durante l'esecuzione di tutte le iterazioni del test. Di default, è utilizzato il nome `values.csv`;

`-t` specifica, in secondi, un limite temporale per l'esecuzione di ogni iterazione del test. Esso ha precedenza rispetto all'argomento `-n`. Di default, è impostato a 0;

`-w` specifica, in secondi, un lasso di tempo di pausa tra l'esecuzione di due diverse iterazioni del test. Di default, è impostato a 5 secondi.

A.6.1 Directory di output

All'interno della directory di output sono presenti due diversi file: `README.txt` e `values.csv` (se non diversamente specificato).

All'interno del file `README.txt` è possibile visionare le impostazioni con cui sono state effettuate le diverse iterazioni del test. Un esempio di file `README.txt` è di seguito riportato:

```
=== vNSF Client Tester - Test settings ===
Concurrency: 50
Keep-alive: False
Time-limit: 0(0 means not used.
    Otherwise, it takes precedence over the number of requests)
Iterations: 3
Waiting time between iterations (in s): 5
Number of requests: 1000
URL: http://localhost:11082/path1/index.html
Output file name (THR in KiB/s): values.csv
Output dir: output_directory
===
```

Il file `values.csv` è un file in formato CSV contenente la lista dei valori di throughput, in KiB/s, registrati durante le diverse iterazioni del test. Ad esempio:

```
1102.47,1171.17,1123.54
```

A.6.2 Esempio di utilizzo

A titolo d'esempio, si supponga di voler ottenere i valori di throughput, in KiB/s, relativi a richieste `GET` di una risorsa `path1/index.html`.

Come punto di partenza, è possibile utilizzare lo script `vNSF_tester.py` descritto nell'Appendice A.4.4. Difatti, per avviare un'istanza locale della vNSF Reverse Proxy con un Origin Server anch'esso locale è possibile utilizzare lo scenario *test1* mediante il seguente comando:

```
[shield@localhost shield-thesis/tests]$ sudo python vNSF_tester.py -t test1
```

In questo modo, si crea un'istanza della vNSF Reverse Proxy che serve un Origin Server locale in esecuzione su un container Docker, raggiungibile all'indirizzo `localhost:11082`.

Si supponga ora di voler effettuare il test con le seguenti impostazioni:

- numero iterazioni: 10;
- grado di concorrenza: 100 utenti;

- numero di richeste: 100000;
- utilizzo di connessioni persistenti;
- tempo di attesa tra due iterazioni: 5 s;
- directory di output: `output_dir`;
- file di output nella directory di output: `valori.csv`.

Il comando da eseguire per effettuare il test è il seguente:

```
[shield@localhost shield-thesis/tests]$ python vNSF_Client_tester.py \
   -i 10 \
   -c 100 \
   -n 100000 \
   -k \
   -w 5 \
   -d "output_dir" \
   -o "valori.csv" \
   http://localhost:11082/path1/index.html
```

All'interno della directory `output_dir` è possibile visionare i file `README.txt` e `valori.csv`. Il contenuto del primo è il seguente:

```
=== vNSF Client Tester - Test settings ===
Concurrency: 100
Keep-alive: True
Time-limit: 0(0 means not used.
Otherwise, it takes precedence over the number of requests)
Iterations: 10
Waiting time between iterations (in s): 5
Number of requests: 100000
URL: http://localhost:11082/path1/index.html
Output file name (THR in KiB/s): valori.csv
Output dir: output_dir
===
```

Il contenuto del secondo, ossia i valori di throughput in KiB/s, sarà analogo a quello di seguito riportato:

```
1785.23,2072.71,2067.81,2066.19,2037.25,2059.56,1822.43,2071.04,2014.38,2023.64
```

Appendice B

Manuale del Programmatore

B.1 Il repository `shield-thesis`

La vNSF Reverse Proxy e il vNSF Controller sono disponibili all'interno del repository https://github.com/vins1993/shield-thesis.

B.1.1 Download del codice sorgente

Per effettuare il download del codice sorgente, è sufficiente clonare il repository mediante `git`:

```
git clone https://github.com/vins1993/shield-thesis.git
```

o procedere al download dell'archivio mediante `wget` o uno strumento analogo:

```
wget https://github.com/vins1993/shield-thesis/archive/master.zip
```

B.1.2 Struttura del repository

Il repository `shield-thesis` presenta la seguente struttura:

```
shield-thesis/
├── collector-docker/
├── mspltranslator/
├── reverseproxy-waf/
├── server_farm_1/
├── tests/
├── vNSF_Controller/
├── configure_etcd_cluster.md
└── README.md
```

La directory `collector-docker` contiene il Dockerfile dell'Immagine `collector_image` e i relativi file necessari per l'operazione di Build.

La direcory `mspltranslator` contiene il codice sorgente del Traduttore MSPL, sviluppato nel presente lavoro di tesi, il Dockerfile dell'Immagine `mspltranslator_image` e i relativi file necessari per l'operazione di Build.

La directory `reverseproxy-waf` contiene il Dockerfile dell'Immagine `reverseproxy_image` e i relativi file necessari per l'operazione di Build.

La directory `server_farm_1` contiene Dockerfile per la creazione di Origin Server da utilizzare per il test in locale della vNSF.

La directory `tests` contiene lo script Python `vNSF_tester.py`, il quale consente di automatizzare l'avvio di una vNSF simulando un comando proveniente da un orchestratore. Al suo interno, sono presenti anche diverse directory che descrivono scenari di test differenti.

La directory `vNSF_Controller` contiene il codice sorgente in Go del vNSF Controller, sviluppato nel corso del presente lavoro di tesi.

Il file `configure_etcd_cluster.md` contiene istruzioni per la creazione di un cluster `etcd`.

Il file `README.md` fornisce una panoramica delle macchine virtuali utilizzate nel corso del presente lavoro di tesi e la loro configurazione.

B.2 Directory `collector-docker`

La directory `collector-docker` presenta la seguente struttura:

```
collector-docker/
├── Dockerfile
├── elk_cluster
├── entrypoint.sh
├── input.conf
├── output.conf
└── update-conf.sh
```

B.2.1 Dockerfile

Dockerfile per la creazione del componente Collettore Log della vNSF Reverse Proxy. Esso prevede l'utilizzo dello script `entrypoiny.sh` come entrypoint del container. L'immagine prodotta include al suo interno i file `input.conf` e `output.conf`.

B.2.2 elk_cluster

Contiene script per l'integrazione del Collettore Log, implementato con Logstash, con istanze di Elasticsearch e Kibana. Il primo si occupa della memorizzazione dell'output di Logstash, mentre il secondo permette di interrogarne il contenuto mediante un'interfaccia grafica. Si veda l'Appendice B.9 per approfondire.

B.2.3 entrypoint.sh

Rappresenta l'entrypoint del Collettore Log. Imposta la variabile d'ambiente `JAVA_HOME` con la directory della JDK desiderata e avvia l'eseguibile `/usr/share/logstash/bin/logstash` all'interno del container.

B.2.4 input.conf

Contiene la configurazione delle sorgenti di Logstash. Essa è stato sviluppata a partire dal progetto https://github.com/bitsofinfo/logstash-modsecurity:

```
input {
  file {
    path => "${AUDIT_LOG_VOLUME_PATH}/*.log"
    type => "mod_security"
    codec => multiline {
      charset => "US-ASCII"
      pattern => "^--[a-fA-F0-9]{8}-Z--$"
```

```
      negate => true
      what => next
    }
  }
}
```

I file sorgenti sono specificati nella variabile `path`, impostata con la directory dell'Audit Log Volume.

Il codec `multiline` consente di inserire più linee del log di un software all'interno dello stesso evento. Nel caso di ModSecurity, tale opzione è indispensabile dal momento che l'Audit Log si presenta suddiviso in più sezioni e in più righe.

Linee appartenenti allo stesso evento sono identificate grazie alla direttiva `pattern`, la quale specifica un'espressione regolare per identificare la chiusura dell'evento. Nel caso di ModSecurity, è noto che la sezione Z è obbligatoria per ogni evento riportato nell'Audit Log. Pertanto, il plugin `file`, considera tutte le stringhe del log come appartenenti allo stesso evento finché non trova una che soddisfi l'espressione regolare riportata nella direttiva `pattern` es. `--5ee9fa00-Z--`. Nello specifico, ogni riga che non soddisfa il `pattern` (`negate => true`) è inserita nell'evento (`what => next`).

B.2.5 output.conf

Contiene la configurazione dell'output di Logstash. Essa prevede l'utilizzo dei plugin `file` e `stdout`:

```
output {
  file {
    path => "/opt/output/out.log"
  }
  stdout {}
}
```

Il plugin `file` consente l'impiego della direttiva `path` per specificare un file di destinazione per le operazioni svolte. In questo caso si è specificato il percorso `/opt/output/` per l'Output Log Volume.

Il plugin `stdout` permette di riportare, nello standard output del container, il risultato delle operazioni di Logstash. Lo standard output può essere letto utilizzando il comando `docker logs` *`container_id`*, dove *`container_id`* rappresenta l'identificativo del container che svolge la funzione di Collettore Log.

B.2.6 update-conf.sh

Contiene uno script Bash per la creazione di un archivio, in formato `tar`, contenente il Dockerfile e i file da esso richiamati. Questo è copiato all'interno delle directory `shield-thesis/vNSF_Controller` e `shield-thesis/tests/vnsf_data/`, in modo tale da poter essere referenziato dai file manifest presenti.

B.3 Directory `mspltranslator`

La directory `mspltranslator` presenta la seguente struttura:

```
mspltranslator
├── vnsfmspl/
├── Dockerfile
└── main.go
```

```
├── modsecurity.go
├── mspl_schema_mod.xsd
├── mspl_schema.xsd
├── mspltranslator
├── README.md
├── template_virtualhost
├── template.go
├── update-conf.sh
├── utils.go
└── xml.go
```

B.3.1 Package `vnsfmspl`

Il package `vnsfmspl` è costituito dai seguenti file:

```
vnsfmspl/
├── interface.go
├── rules.go
├── structs.go
└── types.go
```

interface.go

Il file `interface.go` definisce l'interfaccia `waf`, la quale consente il disaccoppiamento del mspltranslator dall'attuale implementazione della vNSF Reverse Proxy, rendendo il codice portabile.

Per poter tradurre una direttiva in MSPL in una regola comprensibile dal WAF, è sufficiente fornire un'implementazione per il corrispondente metodo presente all'interno dell'interfaccia `waf`.

I metodi attualmente previsti sono:

`Status()` abilita/disabilita l'utilizzo del WAF all'interno della vNSF Reverse Proxy.

`WriteSSLEnabled()` abilita/disabilita l'utilizzo del protocollo SSL per il `<path/>` specificato;

`WriteBlockIP()` indica un indirizzo IP da bloccare tramite l'impiego del WAF per il `<path/>` specificato;

`WriteRestrictPath()` indica un percorso a cui vietare l'accesso tramite l'impiego del WAF per il `<path/>` specificato;

`WriteRequestBodyEnabled()` abilita/disabilita l'ispezione del corpo della richiesta di un Client da parte del WAF;

`WriteRequestBodyLimit()` imposta un limite massimo alla dimensione del corpo di una richiesta HTTP;

`WriteResponseBodyEnabled()` abilita/disabilita l'ispezione del corpo di una risposta di un Origin Server da parte del WAF;

rules.go

Contiene una mappa che collega i pattern di attacco alle apposite regole dell'*OWASP ModSecurity CRS* per contrastarli. La mappa seguente assegna una corrispondenza tra il contenuto degli elementi XML `<pattern/>` della configurazione in MSPL e una lista di file del suddetto progetto:

```
var Rules = map[string][]string{
  "SQL_INJECTION": {"modsecurity_crs_41_sql_injection_attacks.conf"},
  "XSS": {"modsecurity_crs_41_xss_attacks.conf"},
```

```
  "BAD_ROBOTS": {"modsecurity_crs_35_bad_robots.conf"},
  "TROJANS": {"modsecurity_crs_45_trojans.conf"},
  "GENERIC_ATTACKS": {"modsecurity_crs_40_generic_attacks.conf"},
  "OUTBOUND": {"modsecurity_crs_50_outbound.conf"},
  "REQUEST_LIMITS": {"modsecurity_crs_23_request_limits.conf"},
  "TIGHT_SECURITY": {"modsecurity_crs_42_tight_security.conf"},
  "COMMON_EXCEPTIONS": {"modsecurity_crs_47_common_exceptions.conf"},
  "PROTOCOL_ANOMALIES": {"modsecurity_crs_21_protocol_anomalies.conf"},
  "PROTOCOL_VIOLATIONS": {"modsecurity_crs_20_protocol_violations.conf"},
}
```

structs.go

Contiene le strutture dati utilizzate per salvare le direttive estratte dalla configurazione in MSPL.

types.go

Contiene i tipi utilizzati per la scrittura dei template di configurazione.

B.3.2 Dockerfile

Dockerfile per la creazione del componente Traduttore MSPL della vNSF Reverse Proxy. Esso prevede l'inclusione dell'eseguibile `mspltranslator` all'interno dell'Immagine, in modo tale che non sia necessario ricompilarne il codice sorgente.

B.3.3 main.go

Appartiene al package `main` e definisce le funzioni:

`Init()` controlla la validità dei parametri forniti in input dall'utente;

`main()` costituisce l'entrypoint del mspltranslator. Invoca la funzione `ValidateXML()` per la validazione del file XML fornito in input dall'utente, la funzione `ParseXML()` per l'operazione di unmarshalling e la funzione `WriteTemplate()` per la scrittura della configurazione finale.

B.3.4 modsecurity.go

Fornisce un'implementazione dell'interfaccia `waf`, definita nel file `interface.go` all'interno del package `vnsfmspl`.

A titolo d'esempio, si riporta l'implementazione del metodo `Status()` dell'interfaccia `waf`, per l'abilitazione di ModSecurity:

```
func (ModSecurity) Status(value string) string {
  if value == vnsfmspl.ENABLED_VALUE {
    return "On"
  } else if value == vnsfmspl.DETECTION_VALUE {
    return "DetectionOnly"
  } else if value == vnsfmspl.DISABLED_VALUE {
    return "Off"
  }
  return vnsfmspl.DEFAULT
}
```

B.3.5 mspl_schema_mod.xsd

Contiene una versione aggiornata del mspl_schema.xsd, dotata degli elementi XML aggiunti nel corso del presente lavoro di tesi. Essa è utilizzata per la validazione dei file XML utilizzati come input del mspltranslator.

B.3.6 mspl_schema.xsd

Contiene la versione di partenza del mspl_schema.xsd, priva degli elementi XML definiti nel corso del presente lavoro di tesi.

B.3.7 mspltranslator

File binario di mspltranslator. Può essere rigenerato ricompilando il codice sorgente con il comando `go build`, all'interno della directory `mspltranslator`.

B.3.8 template_virtualhost

Rappresenta il template utilizzato per la scrittura della configurazione di httpd con ModSecurity. Contiene sia parti statiche sia parti dinamiche. Le prime permettono di delineare le sezioni della configurazione, mentre le seconde sono segnalate da appositi *placeholder*, ossia delle sequenze di caratteri che indicano una sezione modificabile.

Il template è stato realizzato utilizzando la sintassi del package `text/template` del linguaggio Go:

```
<VirtualHost {{.Rule.SourceAddress}}:{{.Rule.SourcePort}}>
  ServerName {{.Rule.DomainName}}
  {{range $a := .Rule.Paths}}
  <Location {{$a.InputPath}}>
    ProxyPass {{$a.DestinationProtocol}}://\
    {{$a.DestinationAddress}}:{{$a.DestinationPort}}{{$a.DestinationPath}}
    ProxyPassReverse {{$a.DestinationProtocol}}://\
    {{$a.DestinationAddress}}:{{$a.DestinationPort}}{{$a.DestinationPath}}

    #SecuRuleInheritance Off
    SecRuleEngine {{.ApplicationLayer.Action}}
    SecDefaultAction \
    "phase:2,deny,log,status:{{.ApplicationLayer.DefaultResponse}}"

    {{.ApplicationLayer.RequestBodyInspection}}
    {{.ApplicationLayer.RequestBodyLimit}}
    {{.ApplicationLayer.ResponseBodyInspection}}

    #SSL
    #SSLEngine {{.Ssl.Status}}
    #SSLCertificateFile {{.Ssl.CertificatePath}}
    #SSLCertificateKeyFile {{.Ssl.PrivatePath}}

    {{range $b := .ApplicationLayer.Rules}}
    IncludeOptional {{$b}}{{end}}

    {{range $c := .ApplicationLayer.AddedRules}}
    {{$c}}{{end}}
  </Location>{{end}}
</VirtualHost>
```

I placeholder, identificati dai caratteri {{}}, hanno una corrispondenza diretta con i tipi definiti nel file types.go, contenuto all'interno del package vnsfmspl. Per navigare all'interno dei campi delle strutture dati, si utilizza l'operatore ..

B.3.9 template.go

Appartiene al package main e definisce la funzione WriteTemplate(). Essa utilizza il package text/template di Go per la scrittura nei placeholder definiti nel file template_virtualhost.

B.3.10 update-conf.sh

Contiene uno script Bash per la creazione di un archivio, in formato tar, contenente il Dockerfile. Quest'ultimo viene copiato all'interno delle directory shield-thesis/vNSF_Controller e shield-thesis/tests/vnsf_data, in modo tale da poter essere referenziato dai file manifest presenti.

B.3.11 utils.go

Appartiene al package main e definisce le funzioni:

contains() controlla l'esistenza di una stringa all'interno di un array;

close() tenta di chiudere il descrittore di un file lanciando la funzione panic() in caso di errore.

B.3.12 xml.go

Appartiene al package main e definisce le funzioni:

ValidateXML() valida il file XML fornito in input al mspltranslator utilizzando lo schema XML mspl_schema_mod.xsd;

ParseXML() effettua il parsing del file XML dato in input estraendo i parametri da utilizzare per la configurazione del mspltranslator.

B.4 Directory reverseproxy-waf

La directory reverseproxy-waf presenta la seguente struttura:

```
reverseproxy-waf/
├── Dockerfile
└── update-conf.sh
```

B.4.1 Dockerfile

Dockerfile per la creazione del componente Reverse Proxy con WAF della vNSF Reverse Proxy.

B.4.2 update-conf.sh

Contiene uno script Bash per la creazione di un archivio, in formato tar, contenente il Dockerfile. Quest'ultimo viene copiato all'interno delle directory shield-thesis/vNSF_Controller e shield-thesis/tests/vnsf_data, in modo tale da poter essere referenziato dai file manifest presenti.

B.5 Directory `server_farm_1`

La directory `server_farm_1` presenta la seguente struttura:

```
server_farm_1/
└── server_farm_dockerfiles/
    └── httpd_php/
        ├── sample/
        │   ├── private/
        │   │   └── index.html
        │   └── public/
        │       └── index.html
        ├── Dockerfile
        └── README.md
```

B.5.1 Directory `server_farm_dockerfiles`

Contiene una directory per ogni tipo di Origin Server che può essere utilizzato per testare le funzionalità della vNSF Reverse Proxy. Attualmente, è disponibile solo la configurazione di un Server httpd ed è contenuta all'interno della directory `httpd_php`.

Directory `httpd_php`

Contiene un Dockerfile e una cartella `sample`. Il primo può essere utilizzato per il Build di un'Immagine Docker di un Origin Server httpd, mentre la seconda raccoglie pagine di esempio. Queste ultime sono inglobate all'interno dell'Immagine per popolare i contenuti serviti dall'Origin Server.

B.6 Directory `tests`

La directory `tests` presenta la seguente struttura:

```
tests/
├── test1/
├── test2/
├── test2_disabled/
├── test3_host1/
├── test3_host2/
├── test3_host3/
├── test3_loadbalancer/
├── README.md
├── vNSF_Client_tester.py
└── vNSF_tester.py
```

B.6.1 Directory `test1`

Contiene i file `hostconfig`, `manifest` e `mspl.xml`. Essi configurano uno scenario di test con:

- singola istanza della vNSF Reverse Proxy;
- singolo Origin Server in ascolto sullo stesso host;
- abilitazione di tutte le regole di ModSecurity

B.6.2 Directory test2

Contiene i file hostconfig, manifest e mspl.xml. Essi configurano uno scenario di test con:

- singola istanza della vNSF Reverse Proxy;
- singolo Origin Server in ascolto su un host remoto;
- abilitazione di tutte le regole di ModSecurity.

B.6.3 Directory test2_disabled

Contiene i file hostconfig, manifest e mspl.xml. Essi configurano uno scenario di test con:

- singola istanza della vNSF Reverse Proxy;
- singolo Origin Server in ascolto su un host remoto;
- nessuna regola di ModSecurity abilitata.

B.6.4 Directory test3

Contiene i file hostconfig, manifest e mspl.xml. Essi configurano uno scenario di test con:

- tre istanze della vNSF Reverse Proxy;
- singola istanza di Load Balancer;
- singolo Origin Server in ascolto su un host remoto;
- abilitazione di tutte le regole di ModSecurity.

test3_host1

Directory contenente i file hostconfig, manifest e mspl.xml per la configurazione della prima istanza della vNSF Reverse Proxy.

test3_host2

Directory contenente i file hostconfig, manifest e mspl.xml per la configurazione della seconda istanza della vNSF Reverse Proxy.

test3_host3

Directory contenente i file hostconfig, manifest e mspl.xml per la configurazione della terza istanza della vNSF Reverse Proxy.

test3_loadbalancer

Directory contenente i file hostconfig, manifest e mspl.xml per la configurazione del Load Balancer.

B.6.5 vNSF_Client_tester.py

Contiene uno script Python per automatizzare l'avvio di test per il collaudo della vNSF Reverse Proxy secondo le modalità riportate nell'Appendice A.6.

B.6.6 vNSF_tester.py

Contiene uno script Python per automatizzare l'avvio di uno tra gli scenari di test tra quelli succitati. La sintassi richiesta per l'avvio è la seguente:

```
vNSF_tester.py -t <testcase>
```

dove `<testcase>` è sostituito con una delle directory succitate.

B.7 Directory `vNSF_Controller`

La directory `vNSF_Controller` presenta la seguente struttura:

```
vNSF_Controller/
├── CollectorDocker/
├── MSPLTranslatorDocker/
├── ReverseProxyDocker/
├── vnsfmain/
├── hostconfig
├── logger.go
├── main.go
├── manifest
├── README.md
├── vNSF_Controller
└── vnsf_mspl.xml
```

B.7.1 Directory `CollectorDocker`

Contiene un archivio in formato `tar` al cui interno è disponibile il Dockerfile e i file ad esso allegati del Collettore Log. Tale archivio non deve essere modificato manualmente, ma solo tramite lo script `update-conf.sh` contenuto all'interno della directory `shield-thesis/collector-docker`.

La sua presenza consente di testare il funzionamento del vNSF Controller senza ricorrere agli scenari di test descritti nella directory `tests`.

B.7.2 Directory `MSPLTranslatorDocker`

Contiene un archivio in formato `tar` al cui interno è disponibile il Dockerfile e i file ad esso allegati del Traduttore MSPL. Tale archivio non deve essere modificato manualmente, ma solo tramite lo script `update-conf.sh` contenuto all'interno della directory `shield-thesis/mspltranslator`.

La sua presenza consente di testare il funzionamento del vNSF Controller senza ricorrere agli scenari di test descritti nella directory `tests`.

B.7.3 Directory `ReverseProxyDocker`

Contiene un archivio in formato `tar` al cui interno è disponibile il Dockerfile del Reverse Proxy con WAF. Tale archivio non deve essere modificato manualmente, ma solo tramite lo script `update-conf.sh` contenuto all'interno della directory `shield-thesis/reverseproxy-waf`.

La sua presenza consente di testare il funzionamento del vNSF Controller senza ricorrere agli scenari di test descritti nella directory `tests`.

B.7.4 Package `vnsfmain`

Il package `vnsfmain` è costituito dai seguenti file:

```
vnsfmain/
├── Build_image.go
├── Check_Vnsf_Status.go
├── Create_volume.go
├── internal_types.go
├── Launch_container.go
├── main_types.go
├── option_types.go
├── Parse_dockerhost.go
├── Pull_image.go
├── status_type.go
├── Stop_Vnsf.go
├── Vnsf_launcher.go
└── Vnsf_targzip.go
```

Build_image.go

Definisce la funzione `Build_image()` per il Build di un'Immagine Docker a partire da un elemento del vettore `Images` del file manifest.

Check_Vnsf_Status.go

Definisce la funzione `Check_Vnsf_status()` per recuperare lo stato di una vNSF a partire dal file *vNSFid*`_status.json`.

Create_volume.go

Definisce la funzione `Create_volume()` per la creazione di un Volume Docker a partire da un elemento del vettore `Volumes` del file manifest.

internal_types.go

Definisce tipi utilizzati esclusivamente all'interno del vNSF Controller per il passaggio di parametri tra funzioni.

Launch_container.go

Definisce la funzione `Launch_container()` per l'avvio di un container Docker a partire da un elemento del vettore `Containers` del file manifest.

main_types.go

Definisce le strutture dati utilizzate per il *parsing* del file manifest di una vNSF. Il tipo principale, utilizzato per la traduzione del nodo JSON radice, è `Vnsfroot_t`.

option_types.go

Definisce le strutture dati utilizzate per la traduzione delle componenti dinamiche della configurazione, nel file hostconfig. Il tipo principale, utilizzato per la traduzione del nodo JSON radice, è `VnsfOptionsroot_t`.

Parse_dockerhost.go

Definisce la funzione `Parse_dockerhost()` per l'estrazione delle componenti dinamiche della configurazione dal file hostconfig. La stessa funzione aggiorna anche le strutture dati create con la funzione `Parse_vnsfconf()`, definita all'interno del file `Parse_vnsfconf.go`.

Pull_image.go

Definisce la funzione `Pull_image()` per lo scaricamento di un'Immagine Docker dal Docker Hub o dal Registry specificato come argomento nel vNSF Controller.

status_type.go

Definisce le strutture dati utilizzate per la scrittura e la lettura del file *vNSFid*`_status.json`. Il tipo principale, utilizzato per la traduzione del nodo JSON radice, è `Status_t`.

Stop_Vnsf.go

Definisce la funzione `Stop_Vnsf()` per l'arresto di tutti i container appartenenti alla vNSF descritta nel file *vNSFid*`_status.json`. Quest'ultimo è un input della funzione.

Vnsf_launcher.go

Definisce le funzioni:

`Set_Logger()` imposta la variabile globale `vnsflogger`, utilizzata da tutte le altre funzioni all'interno del package `vnsfmain`;

`Vnsf_launcher()` è l'entrypoint per l'avvio di una vNSF. Essa invoca ordinatamente le altre funzioni definite all'interno del package `vnsfmain`.

Vnsf_targzip.go

Definisce la funzione `Extract()` per l'estrazione di archivi compressi in formato `.tar.gz`. È utilizzata per estrarre l'argomento del vNSF Controller.

B.7.5 hostconfig

Contiene un esempio di file hostconfig per la vNSF Reverse Proxy. Esso costituisce il punto di partenza per tutti i casi di test riportati all'interno della directory `tests`.

B.7.6 logger.go

Definisce la funzione `CreateLogger()`. Quest'ultima è tra le prime funzioni richiamate ed è indispensabile per il funzionamento del programma. Se termina con successo, restituisce un puntatore di tipo `*logrus.Logger`, da utilizzare per riportare gli eventi del vNSF Controller. In caso contrario, provoca la terminazione immediata del vNSF Controller con un codice di errore.

B.7.7 main.go

Rappresenta l'entrypoint del vNSF Controller. Definisce le funzioni:

`Init()` controlla la validità dei parametri forniti in input dall'utente;

`main()` invoca la funzione `Init()` e la funzione `Vnsf_launcher()` qualora fosse richiesto l'avvio di una vNSF.

B.7.8 manifest

Contiene la versione attuale del file hostconfig della vNSF Reverse Proxy. Esso costituisce il punto di partenza per tutti i casi di test riportati all'interno della directory `tests`.

```
{
  "Vnsf":{
  "User":"FX132L",
  "Id":"E0BB2F1E8C7A561E9284A141AEDBBE2C",
  "Images":[{
    "Path":"./MSPLTranslatorDocker/MSPLTranslatorDocker.tar",
    "ImageBuildOptions":{
      "Dockerfile":"Dockerfile",
      "SuppressOutput":false,
      "Tags":["mspltranslator_image"]}
    },{
    "Path":"./CollectorDocker/CollectorDocker.tar",
    "ImageBuildOptions":{
      "Dockerfile":"Dockerfile",
      "SuppressOutput":false,
      "Tags":["collector_image"]}
    },{
    "Path":"./ReverseProxyDocker/ReverseProxyDocker.tar",
    "ImageBuildOptions":{
      "Dockerfile":"Dockerfile",
      "SuppressOutput":false,
      "Tags":["reverseproxy_image"]}
  }],
  "Containers":[{
    "Name":"mspl-translator",
    "Config":{
      "Image":"mspltranslator_image",
      "AttachStdin":false,
      "AttachStdout":true,
      "AttachStderr":true,
      "Tty":true,
      "Cmd":["./mspltranslator","-input","/opt/mspl-volume/{{.Mspl}}"],
      "NetworkDisabled":false},
    "HostConfig":{
      "Binds":["MSPL_Volume:/opt/sidecar-docker",
```

```
    "Config_Volume:/opt/proxy-conf:Z"]},
  "NetworkSettings":{},
  "Wait":true,
  "Restart":false,
  "Timeout":30
  },{
  "Name":"reverse-proxy-waf",
  "Config":{
    "AttachStdin":false,
    "AttachStdout":true,
    "AttachStderr":true,
    "Tty":true,
    "Image":"reverseproxy_image",
    "NetworkDisabled":false},
  "HostConfig":{
    "Binds":["Config_Volume:/opt/proxy-conf",
    "Audit_Log_Volume:/var/log/httpd:Z"]},
  "NetworkSettings":{},
  "Wait":false,
  "Restart":true,
  "Timeout":0
  },{
  "Name":"collector",
  "Config":{
    "AttachStdin":false,
    "AttachStdout":true,
    "AttachStderr":true,
    "Tty":true,
    "Image":"collector_image"},
  "HostConfig":{
    "Binds":["Audit_Log_Volume:/opt/input/",
    "Output_Log_Volume:/opt/output/:Z"]},
  "NetworkSettings":{},
  "Wait":false,
  "Restart":true,
  "Timeout":0
}],
"Volumes":[{
  "Name":"MSPL_Volume",
  "Desc":"Sidecar volume with MSPL",
  "Import":["{{.Mspl}}"],
  "VolumesCreateBody":{}
  },{
  "Name":"Config_Volume",
  "Desc":"Reverse Proxy custom configuration from MSPL",
  "Import":[],
  "VolumesCreateBody":{}
  },{
  "Name":"Audit_Log_Volume",
  "Desc":"WAF additional rules",
  "Import":[],
  "VolumesCreateBody":{}
  },{
  "Name":"Output_Log_Volume",
  "Desc":"vnsf logs for aggregation",
  "Import":[],
  "VolumesCreateBody":{}
```

```
    }]
  }
  }
```

In aggiunta alla versione riportata nel Capitolo 5, si nota la presenza delle direttive `AttachStdout`, `AttachStderr` e `Tty`. Esse indicano al vNSF Controller di riportare su un file il contenuto dello standard output e dello standard error dei container. Tale opzione si rivela particolarmente utile in fase di debug della vNSF, mentre è difficilmente utilizzabile in caso di avvio su un ambiente di produzione.

B.7.9 vNSF_controller

File binario del vNSF Controller. Può essere rigenerato ricompilando il codice sorgente con il comando `go build`, all'interno della directory `vNSF_Controller`.

B.7.10 vnsf_mspl.xml

Contiene un esempio di configurazione in MSPL per la vNSF Reverse Proxy. Esso è modificato nelle diverse directory discendenti da `shield-thesis/tests` al fine di creare differenti scenari di test.

B.8 Utilizzo di mspltranslator con diversi Reverse Proxy con WAF

Il codice sorgente di mspltranslator può essere esteso per la scrittura di configurazioni di altri WAF. In altri termini, può essere adattato qualora vi fosse la necessità di modificare l'attuale implementazione del Reverse Proxy con WAF utilizzando una diversa combinazione di Reverse Proxy e WAF.

L'estensione delle funzionalità di mspltranslator dipende da tre fattori: l'interfaccia `waf`, il file `rules.go` e il template di configurazione.

In primo luogo, è necessario implementare i metodi dell'interfaccia `waf`. Attualmente, è disponibile l'implementazione per ModSecurity in `shield-thesis/mspltranslator/modsecurity.go`.

In secondo luogo, nella mappa presente in `rules.go` devono essere indicati file contenenti le regole per la rilevazione dei pattern di attacco. Questi ultimi sono elencati all'interno dello Schema XML `shield-thesis/mspltranslator/mspl_schema_mod.xsd`.

Infine, è necessario fornire un template di configurazione analogo a quello riportato nel file `shield-thesis/mspltranslator/template_virtualhost`. Nello specifico, devono essere individuati dei placeholder per le strutture dati definite nel file `types.go`.

B.9 Integrazione con Elasticsearch e Kibana

Il file `elk_cluster` segnala una porzione di testo da aggiungere al file `input.conf`, utilizzato da Logstash. Il risultato dell'integrazione di tale sezione con l'attuale versione del file `input.conf` è il seguente:

```
output {
  file {
    path => "/opt/output/out.log"
  }
  stdout {}
```

```
    elasticsearch {
      hosts => "192.168.45.34:9200"
    }
  }
```

Si nota l'aggiunta dell'elemento `elasticsearch` alla struttura dati `output`. Esso specifica, nella direttiva `hosts`, l'indirizzo IP e la porta su cui è in ascolto un'istanza di Elasticsearch.

Successivamente, utilizzare il comando per avviare Elasticsearch a partire dalla sua Immagine Docker:

```
docker run --net samplenet --ip 192.168.45.34 \
  -e "discovery.type=single-node" \
  -e "xpack.security.enabled=false"\
  docker.elastic.co/elasticsearch/elasticsearch:5.6.1
```

Si crea così un container in ascolto sulla coppia `192.168.45.34:9200` contenente un'istanza di Elasticsearch.

Infine, utilizzare il comando per avviare Kibana a partire dalla sua Immagine Docker:

```
docker run -p 5601:5601 --net samplenet --ip 192.168.45.35 \
  --add-host="elasticsearch:192.168.45.34" \
  -e "elasticsearch.url=http://192.168.45.34:9200" \
  -e "xpack.security.enabled=false" \
docker.elastic.co/kibana/kibana:5.6.1
```

In questo modo, si avvia un'istanza di Kibana in ascolto sulla porta 5601, già configurata per la connessione ad Elasticsearch all'indirizzo IP `192.168.45.34`. Si noti la presenza di un port mapping, tale da garantire l'accesso al container direttamente dall'host.

B.10 Aggiunta di scenari di test

È possibile aggiungere altri scenari di test, oltre quelli già definiti all'interno della directory `shield-thesis/tests`.

Innanzitutto, è necessario creare una directory, ad esempio `test_sample`. Quest'ultima deve contenere un file manifest, un file hostconfig e una configurazione, secondo il formato specificato nel corso della trattazione.

Inoltre, è necessario apportare modifiche al file `shield-thesis/tests/vNSF_tester.py`. In primo luogo, aggiornare la lista `tests` e il dizionario `tests_desc` aggiungendo il caso `test_sample` e la descrizione del suo utilizzo. In secondo luogo, definire, all'interno della funzione `run_test_case()`, un ulteriore `elif` riportando i comandi specifici per il caso di test che si sta configurando es. creazione di una specifica sottorete.

Infine, se necessario, modificare i parametri con cui si invoca il vNSF Controller.